Efficient Collection and Processing of Cyber Threat Intelligence from Partner Feeds

Martin Kulhavý

School of Science

Thesis submitted for examination
for the degree of Master of Science in Technology.

Espoo, June 19th, 2019

Supervisor

Professor N. Asokan

Advisor

Robin Eriksson, M.A.

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Martin Kulhavý

Title Efficient Collection and Processing of Cyber Threat Intelligence from Partner Feeds

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud Computing **Code of major** SCI3084

Supervisor Professor N. Asokan

Advisor Robin Eriksson, M.A.

Date June 19th, 2019 **Number of pages** 81 **Language** English

Abstract

Sharing of threat intelligence between organizations and companies in the cyber security industry is a crucial part of proactive defense against security threats. Even though some standardization efforts exist, most publishers of cyber security feeds use their own approach and provide data in varying formats, schemata, compression algorithms, through differing APIs etc. This makes every feed unique and complicates their automated collection and processing. Furthermore, the published data may contain a lot of irrelevant records, such as duplicates or data about very exotic files or websites, which are not useful. In this work, we present Feed Automation, a cloud-based system for fully automatic collection and processing of cyber threat intelligence from a variety of online feeds. The system provides two means for reduction of noise in the data: a smart deduplication service based on a sliding window technique, which is able to remove just the duplicates with no important changes in the metadata; and efficient rules, easily configurable by the malware analysts, to remove records which are not useful for us. Additionally, we propose a filtering solution based on machine learning, which is able to predict how useful a record is for our backend systems based on historic data. We demonstrate how this system can help to unify the feed collection, processing, and data noise reduction into one automated system, speeding up development, simplifying maintenance, and reducing the load for the backend systems.

Keywords data feed processing, data deduplication, data filtering, data lake, sliding window, cyber threat intelligence, information exchange, cloud computing, serverless computing, Amazon Web Services, AWS

Preface

The thesis before you presents the work I have done at F-Secure in Helsinki as part of the Feed Automation project. Starting almost from scratch, this project has grown to become the main ingestion point of threat intelligence in the company and it is processing millions of records every day. Developing, deploying, and maintaining such a system has been a challenging, yet rewarding experience, which has taught me a lot. I would like to thank F-Secure for this opportunity to learn and experience how cyber security is done practically in the industry. During the development, I had to communicate with many different teams around the whole company to make sure that the result will match their needs and expectations and during the countless meetings and discussions, I received a lot of help and advice from many fellows from various teams. Although too many to list everyone individually, I want to express my gratitude to all of my F-Secure fellows, who have supported me during this time, your help has been invaluable.

I came to Finland as an exchange student and initially only planned to stay here for 9 months. However, soon I decided to continue my master's studies at Aalto. During my whole time here, I have learned much more than I expected. I am grateful to the Secure Systems Group (SSG) and the Helsinki-Aalto Center for Information Security (HAIC) for strengthening my interest in cyber security and deepening my skills in this field. I was honored with the opportunity to join the Secure Systems Group as a researcher. Working along with the skilled researchers in the group was a very enriching experience.

I would like to sincerely thank my supervisor, Professor N. Asokan, for his patient guidance and support, not just during the course of this thesis, but during my whole studies at Aalto University. His passion for security made a big impact on me and my studies. I am very thankful to my advisor, Robin Eriksson, for his careful and thorough proofreading of my thesis and for his ongoing support during the past months.

Finally, I would like to express my profound gratitude to my family and particularly my parents Lenka and Rudolf, without whose loving support I would never have reached this point. Děkuju vám!

Otaniemi, June 19th, 2019

Martin Kulhavý

Contents

Abbreviations

API	Application programming interface
AUC	Area Under (ROC) Curve
AV	antivirus
AWS	Amazon Web Services
CSV	Comma-separated values
EC2	Amazon Elastic Cloud Compute
ECR	Amazon Elastic Container Registry
ECS	Amazon Elastic Container Service
FRS	F-Secure's File Reputation System
GDPR	General Data Protection Regulation
IAM	AWS Identity and Access Management
JSON	JavaScript Object Notation
KMS	AWS Key Management Service
ML	Machine Learning
MUTE	Malicious URLs Tracking and Exchange group
NATO	North Atlantic Treaty Organization
NRS	F-Secure's Network Reputation System
ORSP	F-Secure's Object Reputation Service Platform
REST	Representational state transfer
RDS	F-Secure's Rapid Detection and Response Service
ROC	Receiver Operating Characteristic (curve)
S3	Amazon Simple Storage Service
SQS	Amazon Simple Queue Service
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privileges
TLS	Transport Layer Security
URL	Uniform Resource Locator, the "web address"
UTF-8	Unicode Transformation Format – 8-bit
VPC	Amazon Virtual Private Cloud
XML	Extensible Markup Language

CHAPTER 1

Introduction

Malware detection in modern antivirus (AV) software is mainly done by comparing the potential malware (its file hashes, static malware signatures, behavior, etc.) with a database of known malware [47]. While some malware might be identified by generic rules, most detections are based on specific malware which was previously encountered and rules carefully engineered by malware analysts to detect that specific malware and its variants. Similarly, when protecting users from browsing malicious websites, the URLs and/or content of visited websites is analyzed and compared against a known database of malicious websites.

Such a *reactive* approach to malware detection requires us to encounter the malware first, *before* the malware can be reliably detected by an AV. Of course, every provider of antivirus protection wants to protect all its customers, including the *first user* who encounters the malware. This can only be accomplished by having information about new malware before it reaches any of the users. This calls for a *proactive* approach – we need to actively collect information about malware from other sources than just the AV users.

This is where threat intelligence comes in. Sharing of information in the cyber security community and industry is of paramount importance. Cyber security researchers and analysts continuously collect information about new threats, learn how they function, how to protect against them, and how to detect them.

Besides the mutual sharing of information among researchers on the Internet or in conferences, there are also automated solutions for information exchange between companies and organizations over the Internet. Various industry partners cooperate and share some of their own intelligence about software and websites. Such metadata may include information whether the described samples are malicious, spy on their users, attempt phishing, include illegal content, etc. Or, conversely, trusted software producers, such as Microsoft, may publish information about their software, which is then whitelisted by AV software, avoiding unnecessary malware detection analysis.

Such information can be used to improve security software, develop new detection rules or improve existing ones, speed up detection engines, or avoid *false positives* (clean samples mistakenly detected as malicious). Thanks to such additional information, users are better protected and cyber security companies can improve their business.

With such strong intelligence exchange, not typical in many other industries, one might ask the question, what motivates these industry players to share their knowledge with others, instead of protecting their valuable know-how? Of course, most companies are profit-oriented and the protection of other companies' users is not necessarily their goal. Therefore different providers and exchange networks use various ways to get value out of this sharing. Some of these data feeds are commercial and only available to users who pay a license fee. Other providers share such information to improve the experience of their own users – for example, it is in the interest of software companies that their users do not experience false detections by AV software, marking their software mistakenly as malicious. Some exchange networks rely on reciprocal sharing – participants can consume data only then if they themselves provide others with good-quality data. Some community-driven feeds are even non-profit and publish the information for free, in an effort to make the Internet a safer place.

Whether paid or free, a lot of data can be collected from many various threat intelligence feeds which are available on the Internet. Of course, not each feed provides enough *useful* information to be worth its costs, both in the license fees and in the amount of work needed to collect and process the data in an automated manner. Each feed needs to be carefully reviewed, whether the provided data is trustworthy and useful.

1.1 Problem overview

Currently, there is no common standard for the cyber security feeds (some efforts for standardized threat intelligence exchange are described in Chapter 8). The data schema, file format, compression algorithm, or API of the feeds are not uniform and so almost every feed is completely unique, making the collection and processing of each feed a very laborious task, requiring a lot of custom work and fine-tuning for each data source.

The feeds provide a lot of data, in some cases millions of records per day (and we expect even higher volumes in the future). However, not all of this data provides useful value. Some feeds contain a large number of duplicates (records referring to the same file or webpage), which do not always contain useful new information and require advanced deduplication techniques. Secondly, not all information in the feeds is always useful for us. For example, some very exotic files will with high probability never be encountered by the company's customers and can therefore be filtered out to conserve computing and storage resources.

1.2 Solution overview

In this work, we will describe the design and implementation of the **Feed Automation** system, which was developed for use in a cyber security company. It can efficiently and automatically collect, combine, and process large amounts of malware-related data collected from various partner sources. We present different methods how the system can find useful information in the data – such methods range from simple rules to sophisticated deduplication and machine learning models. Once all the uninteresting information is removed, the resulting data with new knowledge about malware and malicious websites is submitted to other systems in the company's backend and to the malware researchers and analysts, who use this information to improve our detection mechanisms and ultimately improve the protection of customers.

1.3 Contribution

The presented cloud-based solution shows how to efficiently collect data from many heterogeneous Internet feeds and filter the records to only keep the new and useful information. It shows how such a system can be designed, developed, and deployed to the Amazon Web Services (AWS) cloud. Furthermore, we show that this system can be fully automated, monitored, and fault resistant, based on serverless microservices architecture.

This project has been designed, developed, and deployed at the *F-Secure Corporation* in Helsinki. As a cyber security company developing antivirus software, F-Secure has a big interest in collecting as much useful information about threats as possible, to best protect its customers. The resulting Feed Automation system has been deployed to production and is already providing value to the company. It is an integral component of the new-generation backend system currently being developed at F-Secure as part of the Security Cloud (cf. Section 2.5) and its further development is in planning.

1.4 Structure of the work

In Chapter 2 we give an overview of the problems faced when processing feeds in the cloud, along with information about the situation at F-Secure before the start of this project. In Chapter 3, we present the problem we aim to solve with this work and the goals we hope to reach. Chapter 4 describes the overall system architecture and how we built and deployed this project. In Chapter 5 we go deeper into how we achieved efficient data reduction by deduplication and filtering of records. Chapter 6 discusses the threat model of this project – the possible attack vectors and how we mitigate the risks. In Chapter 7 we evaluate our solution based on the goals we set in Chapter 3. Chapter 8 discusses related work by other authors. Finally, in Chapter 9 we conclude this work and discuss the impact of this project and its possible future improvements.

1.5 Terminology

In this work we describe a project which is processing big datasets of information about malicious websites or files. We refer to such periodically updated datasets as *feeds* and the partner companies or organizations publishing them as *providers* or *publishers*. The information in the feeds is mainly the *metadata* about such objects, such as its maliciousness rating, timestamps when first/last seen, file type, etc. Some publishers might also provide us with the actual binaries of the described files. In this work we generally refer to the actual files or webpages as *samples*. In contrast, we refer to the metadata information about the samples (files or webpages) as *records*.

When discussing data volumes and file sizes, we use the terms gigabyte (GB) and megabyte (MB) to describe $2^{30} = 1,073,741,824$ bytes and $2^{20} = 1,048,576$ bytes, respectively. Some other authors use the binary prefixes and refer to the same units as gibibytes (GiB) or mebibytes (MiB) [51, 59].

CHAPTER 2

Background

In this chapter, we summarize some background information and key concepts which are important to this work. At the end of the chapter, we describe what the initial situation was at the company at the time when this project was started.

2.1 Data Lake

One of the key concepts of this and related works is that of a *data lake*. The term is generally used to describe a centralized scalable repository containing massive amounts of "raw" data, which is readily available to any authorized users or systems for analytics or processing [65]. Some definitions of data lake also include the processing systems which can ingest this data [55]. With raw data we mean data which is in its original form and content as the source data, with minimal alterations. The stored data can be of various types, usually unstructured and arriving in big volumes from other systems.

The data lake has a flat architecture, where each dataset has a unique identifier, optionally with some metadata. However, some categorization is possible. Miloslavskaya et al. [55] describe various separations, for example into 3 tiers: one for raw data, another for augmented daily data sets, and another for third party information. Another possible separation differentiates the data depending on how long should they be stored: data which we want to store indefinitely and data which we want to only store for a limited time, after which it may be discarded to save storage costs.

The concept of data lakes and having all the data in them readily available for use can provide a lot of value to an organization, but many authors also warn of the risk that a data lake may easily become a *data swamp* [65, 42]. Without proper metadata management and catalogizing of the datasets, we may find ourselves in a swamp of data, which nobody can effectively and safely search and use. Various approaches and catalog systems were proposed to help with this problem [19, 42].

Data lakes usually reside in a scalable cloud storage, such as the Microsoft Azure Storage or Amazon S3. Various tools are used to tap into the data, either programmatically with a big data processing engine, such as Apache Spark[1], or with specialized tools with a user interface, such as Splunk[2], allowing access also to users with no or minimal programming knowledge.

2.2 Understanding the feeds and data sources

We collect data from many partners, who provide their data through various APIs and using differing file formats, data schemata, etc. This makes each feed quite unique. In this section, we describe the significant differences between the individual feeds, to highlight the challenge of a unified automated feed processing tool.

The key differences between the feeds are:

- **Update mechanism**. Some feeds are provided as streams, with data flowing continuously, other publish updated packages (for example PhishTank published an hourly dump of *all* known malicious URLs online at that time).
- **Update interval**. Some feeds provide updates at regular intervals, e.g. every minute (such as VirusTotal feeds), once per day (as feeds from AV-TEST), or at irregular intervals (such as the Bing malicious URLs feed by Microsoft).
- The **data volume** ranges from just a few megabytes to tens of gigabytes of uncompressed data per day.
- **Number of records** ranges from just a few thousand to millions per day.
- The average **size of individual records** in the feeds also varies greatly. In Figure 2.1 we show the statistical variation of individual record sizes during one week. We can see that there are big differences. For example, the smallest encountered individual record during that week was only 199 bytes and the largest was 9.1 megabytes.
- The **API** of the feed endpoints is typically a REST API, but might also be an FTP server or a webpage which needs parsing. **Authentication** to the endpoint is done with an API key or a username & password pair, sent in a request header or as part of the URL.
- Data comes in various **file formats**, such as JSON or JSON Lines[3], XML, CSV, or even plain text which has to be parsed. The files are usually **compressed** with zip, bzip2, or gzip compression.

[1] https://spark.apache.org/

[2] http://www.splunk.com/

[3] *JSON Lines* or *newline-delimited JSON* [52] is a text format well suited for processing large datasets one record at a time. JSON Lines consists of a list of JSON values (objects, arrays, etc.), which are separated by a newline character (`\n`). This allows for reading or writing of records line-by-line, instead of reading the complete contents of a file into memory or generating the complete contents in memory before writing it to a file all at once.

- Lastly, the **schema** of the data differs for each feed.

The examples given in the list above are based just on the feeds we have already added to this project, but we can expect more specialties as we keep on adding new feeds. Particularly, in the future we aim to add some feeds with much higher data volume. In Table 2.1 we list some of the feeds, their periodicity, and the average amounts of data they publish daily. Note the large differences between the feeds.

Table 2.1: Example statistics for various partner feeds, based on week 13 of 2019.

Feed name	Periodicity	Average daily uncompressed data volume (in megabytes)	Average daily number of records (in thousands)
AV-TEST Android	daily	0.9	2.9
AV-TEST Windows	daily	17.6	26.7
Bing Malicious URLs	irregular	3,186.5	5,170.0
MUTE Group	continuous	17.4	64.9
OpenPhish	continuous	294.5	589.6
PhishTank	hourly	2,753.7	237.1
VirusTotal file feed	every minute	25,967.0	1,567.8
VirusTotal URL feed	every minute	26,769.2	5,039.1
VirusTotal notifications	continuous	115.4	5.9

In the following, we briefly describe some of the feeds that we concentrated on during the development. This is neither a complete nor a final list of all feeds which this project is supposed to handle, but it should give the reader an overview of the different types of information we are interested in.

- The Germany-based organization *AV-TEST* [38] performs daily evaluations of antivirus software and publishes monthly rankings. They provide subscribers with the samples which were previously used for the testing, both for Windows and Android platforms, along with the results of the tested antivirus programs.
- Microsoft publishes a *Bing Malicious URLs* feed through their Interflow exchange platform [1]. This feed contains URLs which Bing (Microsoft's search engine) identifies as malicious.
- The *Malicious URLs Tracking and Exchange (MUTE)* group [53] is a consortium of companies – such as Avira, Kaspersky, Facebook, or GFI – joined together in an effort to "minimize the exposure of end users from computing threats through timely tracking and exchanging of URLs" [53]. They publish a continuous feed of URLs which were identified as malicious by the submitter (one of the group members).
- *OpenPhish* [58] publishes free or paid feeds of URLs which were detected as phishing websites.

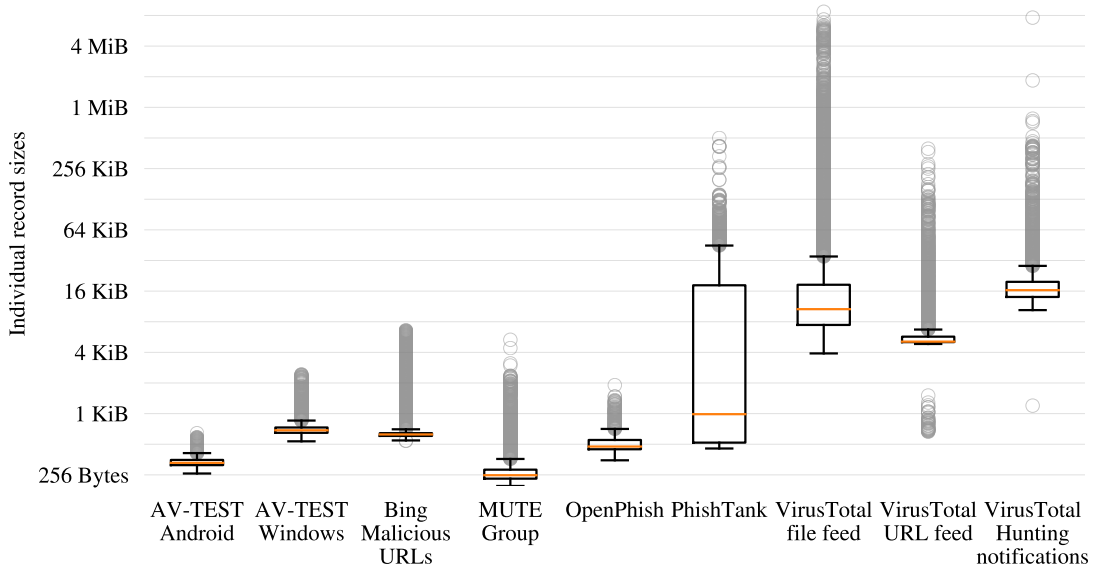


Figure 2.1: Statistical variation of individual record sizes for various partner feeds. In the Tukey boxplots, the box visualizes the 1st, 2nd, and 3rd quartiles, the whiskers denote the lowest and highest values within the 1.5 interquartile range, while the circles denote outliers outside of this range. Note the logarithmic y-axis and the large variations both between the feeds and also within some feeds individually, particularly for the VirusTotal feeds.

- *PhishTank* [57] is a free community-driven website where volunteers can submit, verify, and view information about phishing websites. They publish an hourly-updated archive containing all phishing URLs which are verified and currently online.

- *VirusTotal* [21] is a major aggregator of antivirus software, currently a product of Chronicle (a subsidiary of Alphabet Inc., parent company of Google). Users can submit samples (files or URLs) and collect scan results from various antivirus engines, along with additional metadata extracted by VirusTotal from the sample. Other users can observe the submitted files and URLs and collect scan results, logs and further metadata.

  Besides a powerful web portal, VirusTotal also provides a robust API for automated programmatic access. We are interested in collecting the file and URL live feeds, which contain all the files and URLs which have been submitted to VirusTotal and scanned. Additionally, our researchers are also interested in the feed of notifications from VirusTotal Hunting Livehunt, which can be configured by our own YARA[4] rulesets.

[4] *YARA* (which is an acronym for *YARA: Another Recursive Acronym* or *Yet Another Ridiculous Acronym*) is a tool for detecting malware samples. It allows researchers to define rules based on textual or binary patterns to identify malware in samples. [3, 70]

2.3 Amazon Web Services

Amazon Web Services (AWS) were initially started as a suite of web services for interacting with the Amazon.com online store [48], but soon grew into a major provider of public cloud services and still continues to be the leader in the market [62, 56]. The core AWS services, which we also use in this project, are:

- **Amazon Elastic Cloud Compute (EC2)** for cloud computing, letting users easily setup and use virtual servers [44]. Users can define auto-scaling rules, which deploy extra server instances or shut down already running instances depending on the load or time of day, to adapt to the current amount of traffic. Unlike traditional servers, which need to be provisioned for the highest expected load, auto-scaling can save costs.
  The costs are calculated based on the instance type per instance per hour.
- **Amazon Simple Storage Service (S3)** is an "object storage" [44] which lets users store and retrieve their data from "buckets", identified by string keys. Besides the standard storage class, there are 3 more storage classes for less-frequently accessed data (such as backups) at lower prices.
  The costs are calculated per GB of storage and per request.
- **Amazon Simple Queue Service (SQS)** is a managed message queuing service, which allows for easy communication between distributed software components.
  The costs are calculated based on the number of requests.

Nowadays there are many more AWS services that users can choose from. For example, some of those which we use extensively in this project are Amazon CloudWatch for logs and monitoring, AWS Key Management Service (KMS) for encryption, Amazon Elastic Container Registry (ECR) and Elastic Container Service (ECS) for running Docker containers, or AWS Lambda for serverless functions (see the next Section 2.4). Amazon keeps expanding the AWS portfolio, offering services for machine learning, virtual reality, gaming, robotics, or even managed ground station for satellites control. At the time of writing, the family of Amazon Web Services consisted of a total of 177 services [11].

One of the key characteristics of Amazon Web Services is the pay-as-you-go billing model. As mentioned in the list above, the users only pay for what they use, such as for the exact time spent running a virtual server or the amount of memory used. This is a big difference to the traditional model of owning or renting servers in a data center, where users generally need to allocate (and pay) as many servers as they need during peak time. However, outside of this peak time the servers tend to be underutilized, for example during the night. With virtual servers in the cloud, users can automatically scale the number of server instances up or down as needed at any given moment. This can bring significant savings and also better performance during peak times.

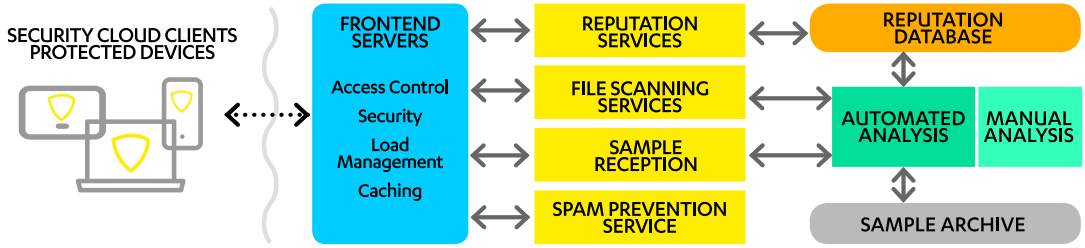


Figure 2.2: Overview of F-Secure Security Cloud. Image source: [26]

2.4 Serverless computing

One of the recent trends in Cloud Computing is the *"serverless computing"* paradigm. The name might be a bit misleading, because underneath the abstraction, servers are still needed to perform the computations, however, the developers do not need to take care of provisioning and configuring the servers themselves [16, 45]. Adzic and Chatley [2] describe serverless computing as "a new generation of Platform-as-a-Service offerings where the infrastructure provider takes responsibility for receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring." In serverless computing, developers have complete control over the application code but have minimal influence on the shared infrastructure.

In practice, this means that developers do not need to concern themselves with server configurations or maintenance, they only need to provide code to handle specific events and configure the needed resources – the cloud provider takes care of the infrastructure, launching the code, responding to requests, auto-scaling, retries after faults, etc. Such serverless event handlers are usually called *functions*, leading to another term *"Function-as-a-Service"* (FaaS) [16]. In AWS, functions are offered as "AWS Lambda" since late 2014 [13].

Due to the event-driven nature of the functions, one significant difference to the usual Platform-as-a-Service offerings is the billing model. With functions, users only pay for the actual time when their code is running, instead of paying for all the time their dedicated server instances are running, including the idle time when the server is just *waiting* for requests.

2.5 F-Secure

F-Secure Corporation is a Finnish cyber security company founded in 1988. Among other products, F-Secure develops antivirus software or endpoint detection and response (EDR) solutions. For all these security products, having a big knowledge base about existing threats is crucial to achieve good protection for customers.

2.5.1 F-Secure Security Cloud

As we discussed in Chapter 1, it is important to keep information about dangerous files or websites. At F-Secure, the knowledge base of digital threats is called the Security Cloud [26, 27]. Under the hood, two subsystems, called the *File Reputation System* (*FRS*) for file samples, and the *Network Reputation System* (*NRS*) for websites, are fed samples and metadata of various files or websites. Upon acquiring this information, these systems analyze it and depending on a complex set of rules and models, they also call multiple different tools and engines to generate as much useful information as possible. This information is then used to generate a verdict whether the given sample is malicious, clean, potentially unwanted[5], etc. Figure 2.2 visualizes how the Security Cloud works. In this diagram, the FRS and NRS systems are part of the "automated analysis" component on the right.

The samples come from various sources – the customers, industry partners or organizations, manually by the researchers and analysts, etc. Although it is desirable to have knowledge about as many samples as possible, the generation and storage of this information is expensive, and so is efficient search over this dataset. Therefore, it is our priority to concentrate only on the samples which were or will be (with certain probability) encountered by the customers and avoid wasting resources on irrelevant samples.

2.5.2 Amazon Web Services and F-Secure

F-Secure started first experiments with Amazon Web Services around 2013. At that time, most of the company software projects were running on company's own servers, either on-premise or in a data center. Around 2016, some projects were already being shifted to AWS – two particular examples are the *Rapid Detection and Response Service* (RDS) [28] and the *Data Pipeline* project, which was later used as a case study to show the rest of the company, that *"we can put our ideas into action faster and save 70% on infrastructure costs by using AWS compared to running hardware on-premises,"* as highlighted by Niina Ojala, the Service Lead [34]. *"We're inspiring a shift toward the cloud, and microservices in particular, because of these great benefits."*

In recent years, F-Secure decided to move completely to AWS, and plans to completely stop using any on-premise services once the transition is complete.

2.5.3 Data collection from feeds before this project

At the start of this project, the feeds collection situation was very fragmented. Various feeds were collected by separate components, scattered around the company in different teams and maintained by different people. Some systems were running

[5] A *potentially unwanted application* or *program* (*PUA*, *POP*) is software which might not be directly malicious (causing damage to programs or data, spreading autonomously, etc.), but may instead perform actions not wanted or expected by the user, such as installing additional unwanted software, spy on user's activities for ads targeting, change configuration of browser, etc. [41].

in the cloud, some on in-house servers, and some even on personal computers. The collected data was also written to various destinations. This approach has many disadvantages in the long term.

- Each time there is a need to start importing a partner feed, it requires considerable development and deployment work. That unnecessarily takes time from developers, while also introducing a higher chance of bugs or holes in security.
- Maintenance of so many different components is tedious and expensive in terms of total time spent by maintainers. The quality of code and documentation differs and if the original author is not available anymore, this can lead to further confusion.
- Different teams might not know about all the data being collected and available for use.
- Data written to various locations leads to complicated permission issues due to different access control imposed on different data sources and different teams. This also makes combining the data from different sources complicated or even unfeasible.
- Applying any advanced functionality, such as deduplication or machine learning models, to *all* imported feeds is near impossible.

Clearly, unifying the process of collecting and processing data from partners brings a lot of benefits, but also a lot of challenges, due to the differences between various data sources and the data they provide.

2.6 Machine learning essentials

In this section, we briefly describe classification using the logistic regression and discuss techniques to evaluate the output and compare different classification models. A detailed introduction to machine learning is out of the scope of this thesis; for more details about machine learning, we refer the reader to literature [39, 60].

In *supervised learning*, the system learns from a dataset comprising of examples that have both input (features) and output values (labels) [60]. Such dataset is called labeled training dataset. It produces either categorical (*classification*) or numeric (*regression*) output.

We can also utilize the numerical output of regression for classification. This is done by using a regression predictor function h with codomain between 0 and 1, i.e. $h(\cdot) \in [0, 1]$. By defining a threshold value, for example $t := 0.5$, we can then use the function h for binary classification, by assigning a sample with features $\boldsymbol{x}$ to category c_1 if $h(\boldsymbol{x}) \geq t$ and to category c_0 otherwise.

Logistic regression model achieves the codomain in range $[0, 1]$ with the logistic function

$$\sigma : \mathbb{R} \to [0, 1] \ : \ z \mapsto \frac{1}{1 + e^{-z}}.$$

The logistic function is an S-shaped monotonic function which takes input from the domain of real numbers and maps it to the $(0, 1)$ range. We can then use the output of a linear regression model in form of $z = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$, where $\beta_0, \ldots, \beta_n$ are the coefficients of the model, and compute the probability p that a feature vector $\boldsymbol{x}$ belongs to the category c_1 as

$$p = P(c_1 \,|\, \boldsymbol{x}) = \frac{1}{1 + e^{-z}} \qquad \text{with } z := \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n.$$

The sample with features $\boldsymbol{x}$ is then assigned to category c_1 if $p \geq t$ and to c_0 otherwise (with e.g. $t := 0.5$).

The performance of a binary classifier can be described by many metrics. In our work, we use the following [60]:

- Accuracy is the fraction of correctly predicted examples by a model.
- True positive rate (*TPR*, also called *recall* or *sensitivity*) is the fraction of positive examples predicted correctly by a model.
- True negative rate (*TNR*, also called *specificity*) is the fraction of negative examples predicted correctly by a model.
- False positive rate (*FPR*) is the fraction of negative examples predicted as positive by a model. It is equal to $1 - TNR$.
- False negative rate (*FNR*) is the fraction of positive examples predicted as negative by a model. It is equal to $1 - TPR$.

In practice, virtually no classifiers achieve perfect 100% accuracy and make some false predictions. When setting up a model, we can influence the balance between the above mentioned rates. For example, by making a model lean more towards the positive category (e.g. by lowering the threshold for logistic regression), we can increase the true positive rate but usually at the cost of lowering the true negative rate at the same time.

A useful way of visualizing this relationship is **receiver operating characteristic (ROC) curve** (Figure 2.3), which visualizes the relationship between the true positive rate (vertical axis) and the false positive rate (horizontal axis) as the threshold is varied. The curve starts at coordinates $(0, 0)$ and finishes at coordinates $(1, 1)$. A perfect classifier (yielding no false positives or false negatives) would be described by a point at coordinates $(0, 1)$, i.e. $FPR = 0$ and $TPR = 1$. Points on the diagonal describe a random guess. Points above the diagonal mean better predictions and points below the diagonal mean predictions worse than random.

When comparing two classifiers, it is useful to compare their ROC curves to see which one can achieve better balance between *FPR* and *TPR*, depending on

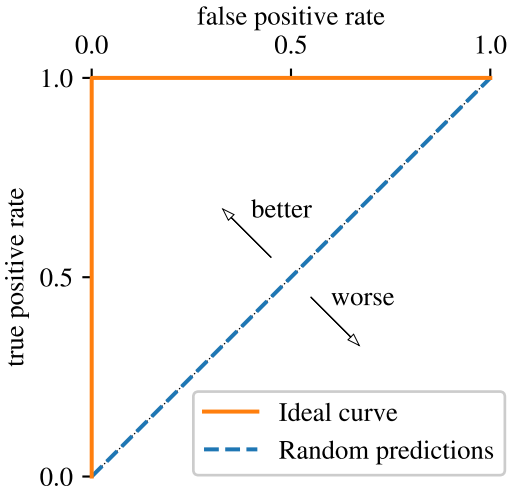


Figure 2.3: Receiver operating characteristic (ROC) curve.

our needs. We can also quantify the overall performance of the classifier across all thresholds with the **area under (ROC) curve** (AUC). The area is computed below the curve in the range $[0, 1]$. Higher scores mean better overall performance than lower scores, with $AUC = 1$ being the best and $AUC = 0$ worst.

2.7 Threat modeling with STRIDE

Microsoft developed the STRIDE model to help their employees identify potential vulnerabilities in their products during a security analysis [49]. It was first formally documented in 1999 and is used by Microsoft for threat modeling of their products [64]. The threats are identified by going through a model of the system, visualized using a data flow diagram (DFD), and for each element asking whether it is susceptible to a threat belonging to one of the following of 6 categories [67].

- Spoofing, when attackers pretend to be someone (or something) else.
- Tampering, when attackers change data in transit or at rest.
- Repudiation, when attackers perform actions that cannot be traced back to them.
- Information disclosure, when attackers steal data in transit or at rest.
- Denial of service, when attackers interrupt a system's legitimate operation.
- Elevation of privilege, when attackers perform actions they are not authorized to perform.

First letters of these categories form the mnemonic STRIDE. After the threats are identified, their mitigations are agreed and documented.

CHAPTER 3

Problem statement

In order to utilize information shared by other partner companies and organizations in the cyber security industry, a system to securely collect and process this data is needed. The providers make this data available in various formats, through various interfaces, released at various intervals. The data provided varies in schema, volume, and targeted content (files, websites, spam campaigns, etc.). We can identify a number of key problems when facing this task.

Complicated maintenance. In general, every single one of these data feeds presents a unique endpoint with unique content and unique issues. The initial simple approach of a separate service to download and process each feed allows for quick development but in the long run, it presents a maintenance challenge to keep all these components running, up-to-date and secure. Furthermore, some code and components are unnecessarily duplicated for multiple feeds.

Scattered deployments. As a consequence of the approach where everyone develops their own service for the feed they are interested in, all these services are deployed in multiple unrelated locations, such as in a public cloud under different accounts, in a private cloud, or even running on personal computers.

Separate processing. While each individual feed provides value, a combination of the feeds can provide additional value, such as the prevalence of certain objects in multiple feeds in a certain time window. However, if each feed is processed completely separately from the others, combining the knowledge is complicated. From a technical perspective, getting data from different locations (private cloud, public cloud, different account, etc.) is problematic and requires extra work and permissions configuration. From an organizational perspective, it is hard to keep an overview about what data is even available to the researchers and analysts, as the information about it is scattered.

Duplication and other data noise. The published data often contains a lot of noise, which is a big problem particularly for the high volume feeds. Without removing

any duplicate or irrelevant records from the feeds, our backend systems might get overloaded processing big amounts of data unnecessarily. However, clearly defining which record is to be considered duplicate or irrelevant for our purposes is not easy. Having a unified solution for all the feeds would greatly reduce the strain on the backend systems where the filtered data are submitted.

During the course of this work, we aim to design, develop, and deploy a unified system, which will centralize the task of downloading data from various feed providers, removing noise from the data, and submitting relevant samples and records for further processing by other backend systems.

It is important that the system workflow is unified, so that data flowing from various providers can share as many components as possible, in order to reduce the development and maintenance costs. However, it is equally important that the system be as flexible as needed for any individual feed. For example, some data feeds need heavy deduplication, while others not at all; some feeds need additional augmentation for each record, while others can pass almost without any alteration.

Finally, as part of the ongoing effort to move most of F-Secure infrastructure and services to Amazon Web Service (cf. Section 2.5.2), the system must be deployed in a cloud environment, utilizing as many serverless components as possible to minimize the operational workload required to deploy and maintain the system.

3.1 Stakeholders

This project will be used by various teams in the company, and communicates with various other systems. To better identify the requirements for the system, we need to narrow down the list of stakeholders, who have an interest in the project. Following is a list of teams which we consider stakeholders of this project. In this list we use descriptive names instead the actual internal team names.

- The **developers and maintainers** are the team that is tasked with the development of the system, its deployment in the cloud, and monitoring, maintenance, and providing support to the other stakeholders.
- The **backend team** maintains the backend components, such as FRS and NRS (cf. Chapter 2), which consume some of the immediate output of this system.
- The **analysts** and **researchers** want to easily search a large knowledge base about malicious threats, to better understand malware behavior and define better detections.
- The **platforms team** administers the base cloud infrastructure, where the whole project will run.

- The **management** and **finances teams** manage the budget and are interested in keeping the overall costs low. They also manage the licenses with outside partners providing us with the feeds.

3.2 System requirements

The main goal of the project, internally called *"**Feed Automation**"*, can be summarized as follows:

> Centralized automated collection, processing, noise reduction, and storage of threat intelligence from various partner feeds.

In cooperation with the various stakeholders of the project, we have narrowed down a list of requirements which the system should meet. Although we do not expect to meet all the criteria in the time given, we hope to meet as many as possible and enabling future additions to expand the capabilities of the system. Below is the list of requirements for the new system.

3.2.1 Centralization

The system should centralize all tasks related to the intake of threat intelligence feeds at the company. Other departments should be able to delegate their feed-importing needs to this system, instead of developing their own services.

Requirement of: developers and maintainers, analysts and researchers.

Evaluation criteria: Requirement satisfied if the system is robust yet flexible enough to take care of all threat intelligence feeds we need to collect and process, without the need for any specialized services outside of this system.

3.2.2 Cloud deployment

The finished project must be deployed completely in the AWS cloud, utilizing serverless architecture wherever possible.

Requirement of: developers and maintainers, platforms team, management.

Evaluation criteria: Requirement satisfied if the system is successfully deployed in AWS cloud, without using any additional infrastructure, such as on-premise servers.

3.2.3 Automation

The system must run autonomously and without the need for any manual input. It must be fault-tolerant and continue working or restart operations even in case of failures.

Requirement of: developers and maintainers.

Evaluation criteria: Requirement satisfied if the deployed system requires no routine manual maintenance to fulfill its day-to-day tasks, even in case of minor faults, such as temporary network outage.

3.2.4 Monitoring

The system must provide monitoring of its health and important metrics and send out alarms in case of failures, such as authentication problems or insufficient received data. The metrics also include the statistics of the feeds.

Requirement of: developers and maintainers, management, analysts and researchers.

Evaluation criteria: Requirement satisfied if the system's dashboards provide an overview of the system health and data flows. The maintainers must be automatically notified about any serious problems with alarms.

3.2.5 Costs

Of course, the overall costs of the system should be reasonably low. To this end, the costs of the system must be easy to monitor.

Requirement of: management and finances team.

Evaluation criteria: Evaluation metric is the overall monthly cost of the system. Requirement satisfied if the complete costs of the system are clearly observable. The goal is that the costs of the system are lower than the costs of previous solutions, while taking into account the savings in the backend systems achieved by reduction of data noise.

3.2.6 Security

Ensure that the data we take in cannot be tampered with and can be trusted. Ensure that the system has no vulnerabilities which could compromise other systems.

Requirement of: all stakeholders.

Evaluation criteria: Requirement satisfied if threat modeling identifies no serious security risks.

3.2.7 Easy management of feeds

Adding a new feed to the system should be quick and require just minimal work, without having to re-implement common functionality or deployment configurations. In the same way, removal of inactive or irrelevant feeds should be quick and easy.

Requirement of: developers and maintainers, analysts and researchers.

Evaluation criteria: Evaluated by developer satisfaction with the process. Requirement satisfied if the developers do not need to spend more than 2 days to add a new feed with an uncomplicated API.

3.2.8 Reduction of data noise

Any records or fields in a record which are irrelevant for us can be considered "data noise" and should be removed. This can be split in three subrequirements:

- **Removal of irrelevant fields** in each record. The system must provide means to modify each record to remove or rearrange fields inside a record.
- **Deduplication** of feed records. A record is considered an irrelevant duplicate if it provides us no new information about a sample. That means we have seen the same record already earlier and no significant information has changed. The system must allow for a fine-grained configuration of which fields in a record are considered significant.
- **Filtering**, which allows both for simple **static rules**, easily editable by analysts and researchers, and also for dynamic techniques, such as filtering based on **machine learning models**.

Requirement of: analysts and researchers, backend team.

Evaluation criteria: Requirement satisfied if the system is able to remove as much data noise as possible while losing as little as possible of any potentially important information. The main metric to evaluate the efficiency of this requirement is the ratio of the amount of data coming into the system from feeds to the amount of data after deduplication to the amount of data being output by the system to the backends. This can be measured both in data volume as bytes and in the number of records.

The minimal goal is to reduce the number of records coming in from the feeds, compared to no data noise reduction at all. On top of this minimal criterion, we would consider it a good achievement if the system is able to reduce the number coming from feeds by at least 50% before submitting them to the backend. This estimated value is based on our initial analysis of the feeds (e.g. the duplicate rates ranging from 1% to 99% for different feeds).

3.2.9 Output

The raw data collected from the feeds should be saved for mid-term storage. The processed data must be saved for long-term storage and accessible. The processed data must be submitted to the appropriate backend systems without unnecessary delay.

Requirement of: analysts and researchers, backend team.

Evaluation criteria: Requirement satisfied if the processed data is continuously saved to data lake, the system experiences no long delays or processing times, and the backend systems are continuously receiving data from the feeds without reporting any faults.
The complete flow of the data from the time it is collected from a publisher until the

time the records are submitted to backend systems should not take longer than 5 minutes on average.

3.2.10 Complete searchability and quick lookup

The system must provide ways for researchers and analysts to search through all the current and historic feeds data in detail. The system should also provide an API for automated systems to quickly gain information about recent feeds data.

Requirement of: analysts and researchers, backend team.

Evaluation criteria: Requirement satisfied if authorized users have easy access to the data for search and analysis.

CHAPTER 4

System design

The main task of the Feed Automation project is to retrieve data from feeds provided to us by remote partner companies or organizations. If new data is available, we want to download and store it in our data lake, and then process it and submit relevant records to our backend systems for analysis. Very simplified, the whole workflow can be described by Algorithm 1.

Algorithm 1: Simplified workflow of the Feed Automation project.

Run this process for each feed at predetermined intervals.

```
1  if new data available at the feed endpoint then
       /* Download the feed data and store it in the data lake          */
2      R_raw ← download_from_endpoint();
3      save_to_data_lake(R_raw);
       /* Process each record, remove duplicates, and store in the data lake
          */
4      R_processed ← {process(r_raw) : r_raw ∈ R_raw};
5      R_processed∧deduplicated ← {r_proc ∈ R_processed : ¬(r_proc is duplicate)};
6      save_to_data_lake(R_processed∧deduplicated);
       /* Apply filtering rules to each record and submit accepted records to
          the backend for analysis                                       */
7      foreach r_proc ∈ R_processed∧deduplicated do
8          if r_proc is accepted by all rules then
9              submit_to_backend(r_proc);
10         end
11     end
12 end
```

Although this whole workflow could be completely deployed inside one application on one server, such monolith applications suffer from many issues [31], such as difficult maintainability, complicated dependencies, technology lock-in, or limited scalability.

Instead, we opted for the microservices paradigm [66], in which we broke up the complexity into multiple smaller services, which perform a limited set of tasks and communicate with each other using messages. This brings multiple advantages:

- Each of the services can be deployed completely separately. The dependencies required are relevant just to the single service. During an upgrade, only the affected service is redeployed, instead of rebooting the whole monolithic system.
- We have greater control over the resources allocated to each service. Some services are memory-intensive, others computationally intensive, while others might spend most time just waiting for external input and do not need an expensive deployment environment. Instead of making compromises for the whole system, we can fine-tune the resources for each service.
- We have complete freedom over what tools, programming language, or platform we choose for each service. Even completely replacing one service for another one can be done very easily, as long as it follows the interface contracts with other services.
- It is often easier to track down bugs, as each service is much more compact and bugs are more localized.

The price we pay for these benefits is a slight maintenance and communication overhead. The contracts between the communicating services need to be precisely defined and enforced to ensure flawless interaction of the services. Detailed metrics, monitoring, and alarms are also needed to ensure that the system is continuously performing as expected.

4.1 System architecture

The system is split into 4 main services:

(**P**) **Pollers and processors** are responsible for polling information from the remote endpoints, downloading new data (if available), processing it, deduplicating the records (by querying the deduplication cache, see below), saving to data lake, and submitting them for filtering.

(**C**) The **deduplication cache** is a service which keeps an overview about what records were recently encountered and in which feeds. It is used mainly by the processors to efficiently deduplicate the records. Its secondary function is providing programmatic access through an API for outside users to lookup the information in the cache. See Section 5.1 for details.

(**F**) The **filtering service** goes through all records that arrive to the filtering queue and applies rules to each record to decide whether to send it further for submission or to drop it (cf. Section 5.2).

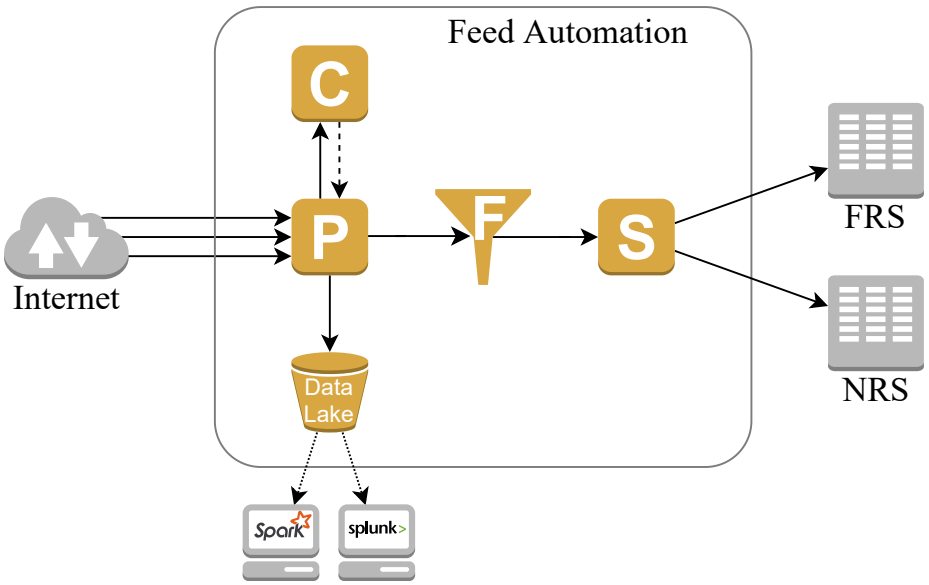


Figure 4.1: Simplified schema of the Feed Automation system and its workflow. Pollers & processors (component **P**) collect data from the Internet feeds and use the deduplication cache (**C**) to remove duplicates. Records are sent both to Data Lake for storage (and analysis using tools like Spark or Splunk) and also through the filter (**F**) to the submitters (**S**), which send them to the appropriate backend (FRS or NRS).

(**S**) **Submitters** collect records from the submission queues and submit them to the corresponding backends, i.e. files to the File Reputation System (FRS) and URLs to the Network Reputation System (NRS) (cf. Chapter 2).

The simplified schema is presented in Figure 4.1. It also shows the flow of data from the remote endpoints, through the pollers and processors (which may query the deduplication cache), and further through the filter to submitters and on to the backend systems. The complete system architecture and workflow is visualized in Appendix A, Figure A.1. In the following sections, we will describe the main components more in detail.

4.2 Pollers and processors

For the downloading and processing of file and website feeds, we use two services, pollers and processors, which have specific separate tasks. For each feed there is a dedicated poller and a dedicated processor service. Figure 4.2 shows the architecture and flow of data between the services.

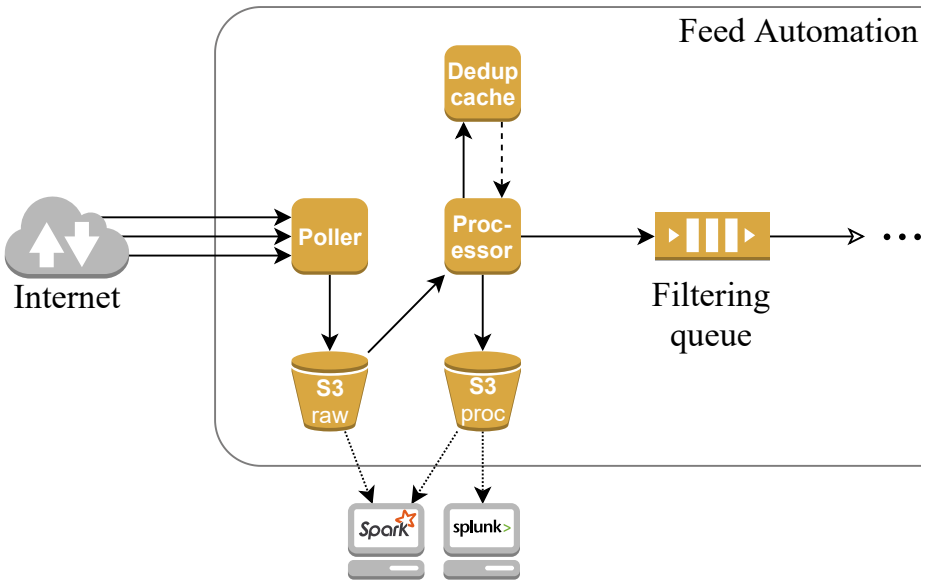


Figure 4.2: Schema of the workflow between pollers, processors, the deduplication cache, and other components.

4.2.1 Pollers

The task of each poller is to query the feed endpoint and check whether there is any new data available since the last downloaded package. If there is (enough) new data, it will download the data package. We call this the "raw" data and want to store it in our data lake. The stored raw data should have as little changes done to it as possible, so that it closely corresponds to the original data we received from the provider. However, the package might arrive in various formats, such as JSON, XML, CSV, etc., optionally compressed with zip, gzip, bzip2 or another compression algorithm. If we would directly store such a package in original format, it would make reading it from data lake complicated, so we want to use a unified format. We opted to store all our data in JSON Lines format, one record per line (encoded as JSON object), compressed with the gzip compression. In case the package already arrives in JSON Lines format, we store it as-is, optionally re-compressing with gzip if the package was not compressed or used a different compression algorithm. In other cases when the package is provided in XML, CSV, or other format, we perform a conversion to JSON; this conversion should not modify or remove any of the information in the data, so that its content and schema resembles the original raw data as closely as possible. When the package is ready in gzipped JSON Lines format, we store it in the data lake. Afterwards, the poller checks whether there is any newer package available. If so, the poller task will be started again.

To correctly ascertain which package is to be downloaded next, the poller needs to store the polling state. This is information about the last downloaded package, such as the package ID, last timestamp, ID of last retrieved record, etc., depending on the feed.

In most cases, it is enough to download the metadata at this stage. However, in some special cases, we may want to also pre-download the file samples before continuing. For example, for the AV-TEST Windows feed, the provider also publishes the corresponding binary samples as one big 7zip archive (tens of gigabytes in size), split into multiple files with maximum size of 512 MB. This whole archive needs to be first downloaded, extracted and stored in the data lake, before we proceed. Because this is a very heavy task, which can take more than an hour, we trigger a separate service[1] to perform the work. Only once the download completes, it triggers the processor.

4.2.2 Processors

Once a new raw package appears in the data lake, a corresponding **processor** service is notified. It will go through all records in the package and process each of them individually. This consists of two main tasks for each record:

1. Process each record. That may include any of the following actions:

   - Removal of unnecessary, superfluous, or duplicate data.
     Some feeds contain fields with dummy or useless information or have the same value for 100% of all records.
   - Addition of new data.
     In some cases we want to add useful information which is available to us during processing but not present in the original record. For example, we might want to add the date of processing or the ID of the source package.
   - Flattening the schema.
     Some feeds have a deep record schema, which is better usable when flattened to just one or two levels instead.
   - Sanitization of the input.
     Some records contain problematic data, such as UTF-8 surrogate pairs, which cause issues to other tools using the output of Feed Automation processors. Such characters need to be replaced or encoded [2].
   - Removal of corrupt or otherwise unusable records.
     If we cannot use the record due to e.g. invalid JSON encoding, it will be dropped already at this stage, before reaching the filtering service.

   The amount of processing strongly depends on the source feed. Records from some feeds require barely any action at all, while others need heavy processing. The main goal of all these actions is to clean or improve the data, while reducing the data size wherever possible to reduce the storage costs.

[1] This is an exceptional service, which is needed only for one specific feed, therefore it is not present in the diagrams displaying the unified workflow.

[2] The Unicode UTF-16 serialization format uses surrogate pairs to encode code points outside of the 16-bit Basic Multilingual Plane (BMP). These surrogate pairs are generally disallowed in other Unicode serialization formats, such as UTF-8.

2. Check whether the record is a duplicate. For this, the processor calculates a special hash for each record, which is calculated from selected fields in the record and represents the record's state. Then it queries the **deduplication cache** about the sample ID (file hash or URL) with the corresponding computed hash. The cache replies with information whether the given record was already seen previously with the given hash. Based on this response, the processor decides whether to drop the record as a duplicate or let it pass. We discuss the deduplication strategy and implementation in detail in Section 5.1.

Both the pollers and processors are services which are very feed-specific. Due to the unique nature of each feed, a specialized poller and processor need to be designed and configured for each. For each new feed poller, the following has to be developed and configured specifically for the given feed:

- Schedule when to trigger the poller. This depends on how frequently the feed is updated. Can be for example once every minute, hourly, daily at specified hour, etc.
- An API client to connect to the endpoint and communicate with it. It must support at least
  - authentication with the endpoint,
  - procedure to gather information about the currently available data, to determine whether (enough) new data is available for downloading, and
  - downloading of the actual feed data.
- A procedure how to compare information about last downloaded package with the newest package available at the feed endpoint.
- A procedure to convert the data package to the desired gzipped JSON Lines format.

For each new feed processor, at least the following needs to be developed and configured:

- A procedure to process each of the records, as described above.
- A procedure to calculate the metadata hash for deduplication (cf. Section 5.1).

It was our design goal to keep all of the feed-specific algorithms and configuration limited just to these two services. That means when a new feed is added to the Feed Automation system, developers only need to create and deploy a poller and a processor for the given feed, without modifying any of the remaining infrastructure.

There are several reasons why pollers and processors are separate components. Besides improved fault tolerance, as we will discuss below, one benefit to such separation of tasks is that we can run the processing in parallel. We do not allow multiple pollers of the same feed to run simultaneously, because this could cause inconsistencies in the order of downloaded data, but the processors, on the other

hand, work on specific packages which were previously downloaded to data lake and can therefore run in parallel if needed. The processing phase is generally more time-demanding than the polling phase, so this is important to avoid data processing lagging behind the data arriving to data lake.

4.2.3 Fault tolerance

As we have seen, the pollers communicate with the remote endpoints in the Internet, yet do not go deep into the content of the data, at most converting the records to JSON. In contrast, the processors do not communicate with any remote endpoints at all, but they do go through the contents of the data record-by-record. This separation of tasks is crucial to achieve robust tolerance to errors, which unavoidably happen when dealing with remote endpoints and data from outside providers.

The most typical error that occurs during a poller's work is a problem in communication with the remote endpoint – either a problem in our proxies, remote server is down, authentication has failed, etc. Thanks to the information we store about the last package which was successfully downloaded, we can easily retry later if the communication with the remote server failed, and continue from the correct package. Many feed providers allow access also to older packages, not just the newest one, so with this approach we do not lose any data. The access to older packages is, however, usually limited, e.g. to the last 24 hours for VirusTotal feeds. If the connection to the feed endpoint is not fixed before this time period, we will start losing data.

The processors face different type of errors, such as malformed data or unexpected schema (missing or superfluous fields etc.). This type of errors are generally not solvable by a repeated execution of the processor, and require a manual fix of the processor code. But thanks to being decoupled from the pollers, a single problematic package causing a processor to fail will not block the whole workflow – the poller will still keep on downloading new data. Secondly, because we have the raw data from the poller stored in the data lake, we can re-run the processor on the problematic package at any time later, after we have improved the processing algorithm.

In practice, the processor faults happen much less frequently than poller faults and after fine-tuning the feed processing algorithm, processor faults occur rarely.

4.2.4 Alternative sources of data

Even though the initial main goal of this project was to collect and process data from feeds, already early in the development we noticed the opportunity to use the Feed Automation system to also process data from other sources. Thanks to the modularity of the architecture, the system is very flexible. Other services collecting data from different sources can work alongside the pollers and processors. They can also optionally utilize the deduplication cache and then they send their output to the filtering queue.

Currently in the deployed system, we use two other components which send data to the Feed Automation (see Figure A.1 in Appendix A):

- Web crawlers, which browse various websites, particularly software portals, for new files to analyze before our customers encounter them.
- Spam collectors, which process emails delivered to special email addresses and collect files and URLs from them.

These components were not developed as part of this thesis work, yet they were already connected into the deployed Feed Automation project and demonstrate its extensibility.

4.3 Deduplication cache and filter

Applying deduplication and filtering to the feeds is the most effective way to reduce the number of records submitted to the backend systems. The **deduplication cache** keeps an overview of which files and URLs were seen recently in the feeds and provides this information to the processors for removing duplicates. The **filter** service allows efficient filtering based on static rules or advanced models. We discuss both the deduplication and filtering in more detail in Chapter 5.

4.4 Submitters

The final stage in the workflow of our system is to submit the records to company backends for analysis or enlarging the knowledge base. By the time a record reaches a submitter, it has been processed, it is not a duplicate, and it has passed all the filtering rules. All that is left is sending it to the appropriate backend. As we described in Chapter 2, the two systems interested in this data are the *File Reputation System* (FRS) for files and the *Network Reputation System* (NRS) for websites. Because their submission APIs are very different, Feed Automation has two different submitters. Each of them has its own queue of records waiting to be submitted.

4.4.1 Submitter to the Network Reputation System

The simpler of the two is submission to NRS, because we submit only the URL and metadata, not the website content. Whenever a new message appears in the URL submission queue, the submitter collects it from the queue, verifies that it contains all the required information, and then sends it to a special REST API responsible for importing data to NRS.

Unfortunately, the current version of the API supports only a very limited set of metadata which we can send, which limits the amount of information we can pass to the backend. This limitation should be relaxed in a future version of our backend systems.

4.4.2 Submitter to the File Reputation System

Submission to the FRS involves more than just sending the metadata information, but also uploading of the binary sample if it is not yet available in the backend. Whenever a new message appears in the file submission queue, the submitter collects it from the queue, verifies that it contains all the required information, and then queries the API of FRS to find out whether the sample is already present in the backend. If it is, the record metadata can be simply sent to the FRS API. Otherwise, we need to download it first. The record from a feed must include a URL where it can be downloaded, either from the Internet, or from our data lake where it was stored at a previous step (e.g. by a web crawler). Once we have the sample ready, it can be uploaded to FRS, along with the metadata.

4.5 Security

Throughout the whole design and development of this system, security has been one of the top priorities. The data output by this system is used by the backend systems as one of the contributing resources to generate verdicts about files and websites, marking them as malicious, clean, potentially unwanted, etc. Therefore we need to make sure that the system is secure and it is especially important that we can trust the data and that nobody can tamper with it. In this section, we will discuss the security measures applied to this system and in Chapter 6 we will again go over the security of the system with threat modeling.

The basic security consideration of this cloud-based system is the security of the underlying infrastructure. Here we rely on the core security guarantees of Amazon Web Services, and trust Amazon to keep our data in storage, running services, and communication between services secured. For example, we make the following assumptions:

- Our user access policies prevent anyone else from accessing our AWS services.
- Our data in Amazon S3 cannot be read or tampered with by unauthorized parties.
- All communication between AWS services is secure and encrypted (using TLS).
- Encryption by the AWS Key Management Service (KMS) is secure.

On top of these basic security assumptions for the infrastructure, here we mention several security measures which we apply to the system.

When configuring the roles and policies for our services using the AWS Identity and Access Management (IAM), we set the permissions to the smallest set needed to perform the service's task. IAM lets us configure fairly fine-grained permissions, so that we can limit what actions and on which resource a service is allowed. For example, a processor service has no need to write to the S3 bucket with raw data,

therefore we only give it reading permission for that particular bucket. Similarly, only the filtering service is allowed to receive and delete messages from the filtering SQS queue and it is also the only service allowed to send messages to the submission queues.

Most of the APIs we communicate with (both feed APIs and backend APIs) require API keys or username and password authentication. To secure these credentials and avoid their accidental leak, we store them in the AWS Secrets Manager, additionally encrypted by the AWS Key Management Service (KMS).

It is crucial that we can trust the feed data we receive from our partners. First and foremost, we need to trust the partner, so the following must be asserted:

- The data produced by the partner is reliable and of high quality,
- the partner's systems are secure from tampering, and
- the partner immediately informs us of any breach of security which could affect the integrity of the data.

We cannot guarantee the above requirements by technical means in this system. It is the task of the team managing the licensing contracts with our partners must make sure that they apply during the whole period when we use the data from any partner. In our system, we must ensure that any downloaded feed data came from the partner without being tampered with. We accomplish this by using the Transport Layer Security (TLS) protocol for all connections to any API and verifying the certificate.

4.6 Implementation

The Feed Automation system has been deployed in Amazon cloud, utilizing many of the provided services. The whole data lake is stored in Amazon S3 storage, using one S3 bucket for the raw data (as generated by the pollers) and another S3 bucket for the processed data (as generated by the processors).

All of the AWS resources are deployed in a dedicated and isolated network, using the Amazon Virtual Private Cloud (VPC). For improved security, the network does not have access to the Internet or any outside services, except through special proxies.

We used the serverless AWS Lambda functions for most of our computing components, wherever possible. The advantages which we described in Section 2.4 make serverless functions ideal for use in our project. The feeds are updated at various intervals from minutes to hours or days (cf. Table 2.1). It would be wasteful to keep (and pay for) a server running unnecessarily idle between the requests. Serverless functions allow us to start them only at specific intervals or by a custom trigger (such as whenever a new file appears in data lake), pay for a few seconds or minutes to perform their task and terminate.

However, the AWS Lambda functions also have certain limitations. Namely, the maximum execution time is limited to 15 minutes (the limit used to be only 5

minutes until October 2018 [7]), maximum memory is 3,008 MB, the file system has only 512 MB of disk space available to use [8], etc. Due to such limits, AWS Lambda is not well suited for heavy tasks, such as downloads of massive archives or uploads of big files. In such cases, we used a more traditional solution of Docker containers running on Amazon Elastic Compute Cloud (EC2) server instances using the Amazon Elastic Container Service (ECS)[3]. The used AWS components are listed in Table 4.1.

Table 4.1: Feed Automation services and the corresponding AWS computing platform used for implementation.

Service	AWS service	Configuration
Pollers	Lambda	1024–2048 MB of memory
Processors	Lambda	1024–3008 MB of memory
Filter	Lambda	1024 MB of memory
Submitter to FRS	ECS	2× c5.xlarge EC2 instance*
Submitter to NRS	Lambda	1024 MB of memory
Deduplication cache API	Lambda	1024 MB of memory
AV-TEST downloader	ECS	2× c5.xlarge EC2 instance*

(*) The EC2 instances of the ECS cluster are shared by all ECS containers in the Feed Automation project, including the crawlers and spam components.

When configuring the computing components, we need to balance the performance and the price. AWS Lambda functions are configured by specifying the allocable memory (in MB). This setting determines not only the maximum amount of memory that the function can use, but also the CPU power and the price per 100ms – doubling the memory setting will also double the CPU power and also double the price (cf. current AWS Lambda pricing schema in Appendix C, Table C.3). In practice, this can mean that doubling the memory setting of a computationally intensive Lambda function will make the function finish in half the time, incurring the same costs as previously! On the other hand, if the function spends most of its running time waiting for input (such as downloading a big file from a slow server), increasing the setting will not make the function run much faster, yet it will cost more. In our case, the separation of pollers and processors fits this well, because pollers are input-intensive and processors are computationally intensive, so we generally configured them with lower and higher memory settings, respectively.

The communication between the individual services occurs mostly through messages passed via queues (Amazon Simple Queue Service, SQS). Queues help to spread the load according to the throughput of consuming service – the messages

[3]Amazon also offers a "serverless" solution for ECS, called *Fargate*. It simplifies the launching of containers without the need to provision and configure EC2 instances (although whether it can be considered truly "serverless" is disputed [71]). Unfortunately, due to technical limitations of Fargate combined with some company-internal AWS deployment policies, we are unable to use it for this project at this time.

are processed as quickly as the consumer can process them. Additionally, queues improve the fault-tolerance – failed messages are not deleted from the queue and are automatically retried later.

There are two exceptions which do not use queues for communication. First is the triggering of processors. Instead of the poller sending a message to the processor through a queue, we use the event notification feature of Amazon S3. Whenever a new feed package is saved to the data lake's S3 bucket, a processor for the corresponding feed is triggered and instructed to process that package.

The second exception is the communication between processors and the deduplication cache. As we will discuss in detail in Section 5.1, the deduplication cache service consists of a Redis cache and a Lambda function which responds to the deduplication queries. These queries are synchronous, awaiting a response. To achieve that, the processor invokes the cache Lambda function with a special `RequestResponse` invocation type, which keeps the connection open until a response from the cache is sent back.

4.6.1 Monitoring

When maintaining a complex system built up from microservices, it is crucial to set up thorough monitoring. Without observing a wide range of metrics, it is difficult to assess the system's health and performance.

In AWS, metrics are made available through the Amazon CloudWatch service. There are many built-in metrics for each AWS service, for example the number of errors or function running times for AWS Lambda[9], the number of waiting messages in a queue for Amazon SQS [6], or the number of evicted keys for Amazon ElastiCache for Redis [12]. Such metrics are already available without any configuration needed, but additionally, we can also log our own metrics, such as the number of records flowing through.

In the following list we show a selection of some of the most important metrics we collect and observe for our services (note that this is not a complete list).

- For each poller:
  - amount of data downloaded from feed endpoint (both compressed and decompressed, if available), and
  - amount of data stored to S3 (in bytes).
- For each processor:
  - number of records received,
  - number of records dropped as duplicates, and
  - number of records dropped for other reasons.
- For the deduplication cache:

  - number of records received,
  - number of records identified as “new”,
  - number of records identified as “modified”,
  - number of records identified as “seen”,
  - memory used by each of the Redis caches, and
  - number of records evicted once the memory is full.
- For the filter service:
  - number of records received, and
  - number of records output (not dropped).
- For submitter services:
  - number of records submitted to FRS/NRS backends.
- For each Lambda function:
  - number of errors (e.g. when the function timed out, ran out of memory, or terminated unexpectedly due to an exception),
  - number of function invocations, and
  - function running time.
- For each SQS queue:
  - number of messages waiting to be received,
  - number of messages being processed (received but not yet deleted), and
  - age of the oldest message waiting.

All metrics can be observed either directly in the Amazon CloudWatch web console or through an external tool (we use mainly the Grafana[4] tool). They are very useful for understanding how the system is behaving or looking for irregularities. They can also be used to set up alarms, which can actively notify the maintainers of the system of any serious problems by sending an email or text message. For example, we have alarms configured to notify us whenever too many Lambda errors occur or when the pollers are collecting insufficient amounts of data, which usually signals problems with our Internet connection.

For a more detailed monitoring of the service’s activity, Amazon CloudWatch also provides a Logs service, which allows us to observe the logs which were output by the Lambda functions or ECS container tasks. This is useful for debugging the services and following their activity in detail.

[4] https://grafana.com/

CHAPTER 5

Data processing and filtering

One of the main tasks of our project is to reduce the load on our backend processing systems. To achieve that, we want to avoid submitting all of the data which came from the feeds to the backend. Instead, we want to remove as many records as we can, while retaining as much cumulative information as possible. Records with information that is of no use to our customers (such as very exotic files), or with information that we already have, brings us no additional value, yet incurs processing costs. Such data can be categorized as data noise, and should be dropped.

5.1 Data deduplication

The first step in reducing the number of records in a big data system is to look for possible duplicates and determine whether the information in the record is already present in our knowledge base. This can be achieved either by directly querying our backend systems, or by maintaining this overview in our system. Because the former approach would incur additional load on the backend systems, which we are trying to lower, we use the latter approach.

5.1.1 Initial version

In our initial efforts, we concentrated on making the system very light-weight and simple. Our deduplication algorithm was included directly in the feed processor and operated statelessly on each received batch. Depending on the source feed, we waited until we accumulate enough data (e.g. half-hour of VirusTotal URL feed) and performed processing and deduplication on each batch. We have based the deduplication on the URLs for websites and the SHA-256 hashes for file records. The advantage of this approach is its simplicity. It is entirely stateless, requiring no extra storage or additional computational power, therefore keeping the costs low. However, it has two significant disadvantages. First, because we are batching the incoming data, the system has to wait until enough data has been accumulated – this

causes delays in the delivery of the data to backend. Secondly, we are only detecting duplicates inside each batch but not duplicates across different batches – this makes the deduplication less effective over longer periods of time.

5.1.2 Stateful deduplication

Ultimately, we resolved to try a more robust stateful approach to deduplication. To keep the state, we use a sliding window technique [20] to store knowledge about recent records. At any given point in time, the window contains information about what records have appeared in observed feeds in recent time. A visualization of this technique is shown in Figure 5.1.

For example, if we start with an empty window and encounter a file F_1, we store its hash to the window and process the file. Once we encounter another file F_2, we look in the window whether we have seen it recently. Because we did not, its hash is stored in the window and the file is processed. If we encounter file F_1 again, we can see that it is already present in the window. That means this file is a duplicate and it can be dropped.

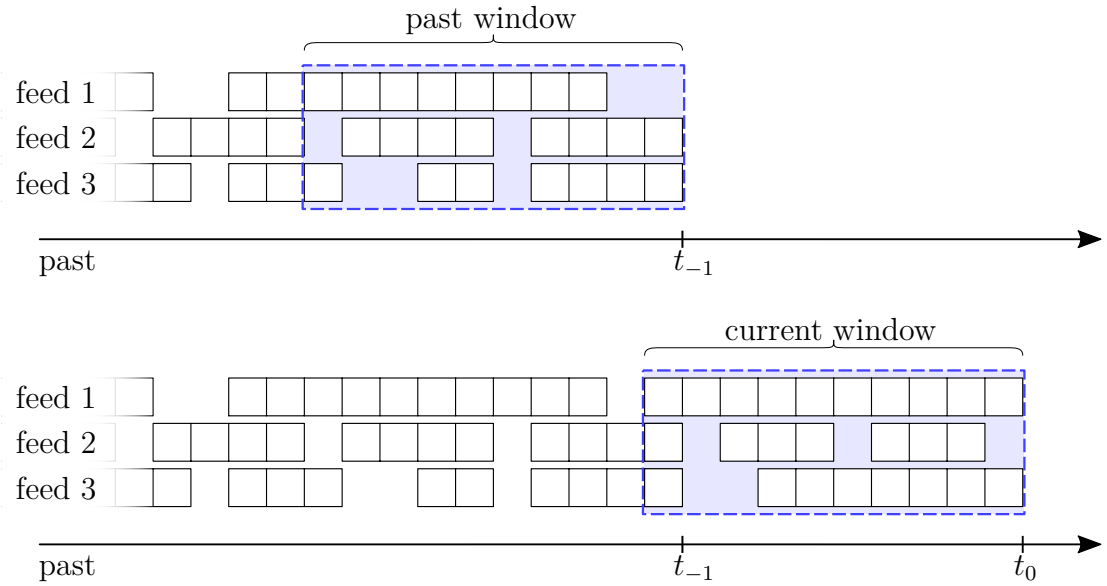


Figure 5.1: Visualization of the sliding window technique for the deduplication of three feeds. In the top graphic is a past window of knowledge at time t_{-1}; while in the bottom graphic we see the current window of knowledge at time t_0.

We can also store additional information, such as when this record was seen for the first time, when it was last seen, how many times, etc. Based on this information, the deduplication logic decides whether a newly appeared record is considered a duplicate or not and consequently whether it should be accepted for further processing or dropped.

Information about the files remain in the window until it becomes full, then the oldest records will be discarded. The size of the window is limited only by the size of storage we have available (instead of fixing the maximum number of records or age of oldest record). The size of the window must be chosen so that it balances the number of records we want to keep in the window and the running costs of the storage. With more memory, we can store more records and see further into the past, but this brings higher running costs.

5.1.3 "Smart" deduplication

The most basic form of deduplication compares only the record identifiers, i.e. the URLs of web pages or the SHA-256 hashes of files. However, generally we are interested in important changes of the metadata that we collect from our partner sources. For example, if we encounter the same record for the second time, but its rating has changed from "clean" to "malicious", we do not want to drop this information as duplicate, but instead let our backend systems know about this important change. Therefore, we need to take certain additional metadata into account during deduplication, so that only records with no new important information are dropped as duplicates.

As an example, let us consider records describing files, with attributes `sha256`, `last_seen`, and `malicious`. In a feed, we encounter the following 3 records:

record1
- `sha256`: 012..def
- `last_seen`: 12:00
- `malicious`: *false*

record2
- `sha256`: 012..def
- `last_seen`: 13:00
- `malicious`: *false*

record3
- `sha256`: 012..def
- `last_seen`: 14:00
- `malicious`: *true*

All three records describe the same file with identical SHA-256 hash. A basic deduplication approach based just on the hash identifier would only retain just the first record and drop the following two as duplicates. However, with our "smart" approach we can specify that we want to know if the important `malicious` attribute has changed, but at the same time we do not care about changes of the `last_seen` attribute. Applied to the three records above, only `record2` would be dropped as a duplicate, as only the value of `last_seen` has changed, but `record3` would be retained, because the important `malicious` flag has changed.

For efficient storage in the deduplication window, we keep only a hash of this metadata, instead of storing all the data. This is enough to detect any changes in any of the selected metadata.

5.1.4 Implementation

The deduplication service consists of two main components. One is a high-performance storage, which maintains the current state of the deduplication window. The second is an interface which provides an abstraction layer over the cache, acting as an API responding to requests from other services, such as the feed processors.

To achieve high performance of the deduplication storage, we chose Redis, which is an in-memory key-value data store. Thanks to keeping all the data in main memory, instead of reading from disk, it is well suited for high-performance caching, which makes it ideal for the implementation of our deduplication sliding window. In AWS, Redis is available as part of the ElastiCache service. We chose to keep two separate Redis instances, one for files and one for URLs.

In Redis, data is stored as a `key=object` pairs and various object types are supported [50]. In our implementation, we use the object's identifier as the Redis key and a hash map for values to store the metadata we need. The ideal schema for our data would be the following:

- `01234567..89abcdef` ..... identifier
  - feed A
    - seen counter ..... how many times was this record seen in feed$_n$
    - metadata hash
    - first seen timestamp
    - last seen timestamp
  - feed B
    - ...
  - ...
- ...

Unfortunately, Redis does not support nested hash maps, so it is not possible to directly use the above schema. Instead, it is necessary to mimic nested hash maps by encoding the data in the keys or objects. To achieve that, we have investigated three variants:

(i) Incorporate the feed identifier into the Redis key.

- `01234567..89abcdef:feed`A
  - seen counter
  - metadata hash
  - first seen timestamp
  - last seen timestamp
- `01234567..89abcdef:feed`B
  - ...

(ii) Incorporate the feed identifier into the hash field name for each field

- `0123456789abcdef`
  - feed$A$:seen counter
  - feed$A$:metadata hash
  - feed$A$:first seen timestamp
  - feed$A$:last seen timestamp
- ...

(iii) Save all metadata in an encoded string, such as JSON.

- `0123456789abcdef`
  - feed*A*={"seen_counter": *number*, "metadata": "*hash*", "first_seen": *timestamp*, "last_seen": *timestamp*}

The first approach provides a clean hash schema for each "identifier–feed" pair and we can also specify expiration time for each. However, listing all encountered feeds for a given record is very inefficient – Redis is quick to retrieve a value for a given key, but slow to list keys by prefix. Furthermore, the identifier, which may be very long (such as a long URL), has to be stored multiple times, unnecessarily wasting storage space.

The second approach allows easier iteration over all feeds where the object was seen, thanks to efficient retrieval of subkeys of a given hash map. We can only set expiration on the whole key, therefore the individual feed records do not expire automatically.

Finally, the third solution wastes space and limits the actions we can directly perform in Redis with individual fields.

Given the facts described above, we selected the schema (ii).

The size of the sliding window is determined by the size of the available Redis memory. However, it is impossible to precisely predict how many records will fit into the sliding window, as this depends on the size of data stored in each key. For example, if file F_1 appeared in just one feed, it will take considerably less space than a file F_2 which appeared in five feeds. Due to the structure selected above, both will count as one key, yet consume very different amounts of memory. Neither is it possible to predict how long the time frame of the deduplication window is, as this also depends on the amount of new records coming in.

In Table 5.1 we present some values from our testing, which provide a good guiding value for how much can fit into the deduplication cache. These can be used when deciding on the optimal Redis instance type. The AWS ElastiCache costs for big memory instances are substantial (cf. Appendix C, Table C.2) and make a very significant part of the Feed Automation budget. Finding an optimal size for the deduplication cache is therefore important to balance the benefits of efficient deduplication and the corresponding costs.

5.2 Filtering

After the deduplication phase, the second step to reduce the number of records is to perform filtering of the data. We differentiate two types of filtering:

- Static rules to determine which records should be accepted and which records

Table 5.1: Redis memory statistics for a fully filled deduplication cache.

Record type	ElastiCache memory (in gigabytes)	Memory allocated (in gigabytes)*	Keys count (in millions)	Memory allocated per key (average, in bytes)
URLs	28.4	21.3	62.1	368.3
URLs	6.1	4.5	13.1	337.6
files	13.5	10.1	32.1	372.2
files	2.8	2.1	6.6	341.9

(*) Memory allocated is the total memory used for storing the actual data, without the overhead.

should be dropped, based on the values of the fields of each single record.
For example: drop every record whose last modification date is older than a month.

- Dynamic filtering based on machine learning models, which adapt and evolve over time as our knowledge of the data increases.
For example: drop those records which with high certainty are not interesting to us, based on previous experience.

Unlike deduplication, which only removes records we have already seen and stored, filtering removes even records which are potentially useful. Therefore it is crucial to properly define and understand the filtering rules, so that only the irrelevant records, the "noise", are filtered out, otherwise we may lose useful information.

5.2.1 Static filtering with rules

In some cases we can quickly determine that certain records are surely not useful to us and can be dropped without losing any important information. Static rules are ideal for this scenario, as they are simple to define, easy to understand, and quick to evaluate.

There are various rule engines available to define complex sets of rules. One of the early systems is CLIPS (*"C Language Integrated Production System"*), which is considered to be probably the most widely used expert system tool [30]. The more modern rule engines include *Jesse* or *DROOLS*, both written in Java. Our filtering component (as described in Chapter 4) has been purposefully designed as a stand-alone component, allowing the use of any rule engine on any platform, which can read and output SQS messages. This allows for easy "in-place" swap with another component based on a different rule engine if needed.

As we currently do not expect any need for complicated rulesets, we have opted for a simple Python approach. Such a solution is very lightweight, without the need of any specialized rule engine, and easy to understand, as the Python syntax is more widely understood than the specialized rule syntax of e.g. CLIPS or DROOLS.

One of our main aims for these rulesets is that they need to be very easily understandable and editable by any analyst, without specialized knowledge of the filtering component itself. We achieved this by providing a very simple rule interface, which consists only of a Python function returning either `True` or `False`, based on the fields of the record. The scope of the rule can be narrowed down with a function decorator[1] to specific feeds.

For example, in Listing 5.1 we can see a rule which applies only to the VirusTotal URL feed, which instructs the filter to drop any records which have zero positives (the number of vendors claiming the given URL is malicious) and no categories specified. Such records are currently not useful for our backend systems, therefore we have no need to process them further.

Code Listing 5.1: An example of a filtering rule for VirusTotal URL feed.

```
@feeds('virustotal-url')
def virustotal_url_is_url_accepted(positives, categories, **fields):
    """Drop URLs with zero positives and no categories.

    :param positives: number of positive verdicts by vendors, as reported by VT
    :param categories: categories information, as reported by VirusTotal
    :param fields: dictionary of all remaining fields available in the record
    :return: True to accept the record, False to drop it
    """
    return positives > 0 or categories
```

As the example shows, the rule consists just of three easily understandable lines of code and helpful comment[2]. In practice, this allows the malware analysts to quickly understand and adapt the rules with minimal assistance from the software engineers or need to learn any new rules language.

5.2.2 Dynamic filtering

Instead of just static filtering rules, which base their decision solely on the contents of a single record, we may also use advanced techniques, which learn from older records we have seen earlier and provide us with a filtering algorithm that is continuously adapting to the incoming data we have seen. For example, we can achieve more advanced, intelligent filtering with machine learning (ML) by learning from the historical feed data to predict which records would be useful or "interesting" to our malware analysts or backend systems. Such records might be those referring to malicious files or phishing URLs, but also records which could be marked as false positives or have a high probability of being encountered by our customers.

[1] In Python, a decorator is any callable which can be used to modify the behavior of another wrapped function, method or class. A function is "decorated" by adding `@decorator_name(**optional_params)` above its definition. Python decorators are similar to annotations in Java.

[2] The docstring we use to document the rule function is in the reStructuredText format. It describes the function, its parameters, and the return value.

To this end, we explored the possibility of using supervised learning to train a model which could predict which records are "interesting" to us. In our case, the input data are the records coming from each individual feed. The output values (the *labels*) describe how "interesting" a given record is. Categorical labels might be just binary "interesting" and "uninteresting", or even "malware", "potentially unwanted", "clean", etc. At this point, it is important to stress that the goal of this system is *not* to generate verdicts about any samples, only decide what records will be imported and thus which samples will get analyzed at this stage. However, the results are used by other systems, which might adapt their verdict or analysis workflow based on this (and other) data. For that reason we need to be careful not to try to guess the verdict and rather use the binary "interesting" / "uninteresting" classification.

The second option is to use regression to generate numeric output. Such a number can be interpreted as an index or probability of how interesting would a given record be. Then we can define a threshold, under which we consider records "uninteresting" and above "interesting". The advantage of such approach is that we can adapt this threshold as needed even without retraining the model.

Whichever strategy we select, the goal remains to only drop records which are not interesting with a high probability. In other words, false positives (uninteresting records which we accept and submit needlessly) are less of a problem than false negatives (dropping interesting records and not submitting them). False positives cause unnecessary computing and storage costs for our backend systems, while false negatives cause us to lose potentially useful information.

Experiment

Dynamic filtering is currently not deployed as part of the Feed Automation system. However, we explored possible ML-based filtering strategies and here we present one experiment which serves as a proof of concept for the use of dynamic filtering on feeds with threat intelligence.

It is clear that we cannot apply the ML approach to every feed. Most of the feeds we currently collect do not contain many fields which would be particularly useful as features for the ML process. Secondly, the typical problem in supervised learning is getting a training dataset which is big enough, is a truly random sample, and has good quality labels. In our case we need labels categorizing records as interesting or not interesting, which is hard to get.

Based on these considerations, we selected the VirusTotal file and url feeds for our experiment. The VirusTotal feeds consists of reports for files and urls which were submitted to VirusTotal and contain various information. As an example, in Appendix B the Listing B.1 shows one file report for an EICAR test file [14]. The content of the report varies depending on the sample (we have shown the significant differences in sizes of individual records in Figure 2.1), but all the reports contain information about scan results from the same vendors (although the list of vendors can change slightly over time). As we can see in the example, scan result for each

vendor tells us whether the sample was detected by the vendor, results of the detection, date of the scan and version of the scanner. The detection result contains the name of the detection as given by the vendor and is generally not compatible between the vendors. Similarly, the VirusTotal URL feed contains scan results from various vendors about the maliciousness of webpages.

As features for our model, we use only the binary results *detected* (1), or *not detected* (0) by each vendor. As labels, we use the verdicts given by our internal Object Reputation Service Platform (ORSP) [27]. This backend service provides clients with information whether any queried sample (file or URL) is categorized as malicious, unknown, clean, etc. For our filtering scenario, that means we only consider the maliciousness for how "interesting" a sample is; or, in other words, we try to predict whether ORSP would declare a sample as malicious (1) and or not malicious (0). In future research, we plan to also explore other hypotheses exploring e.g. the prevalence of a sample or its categorization (adult content, gambling, etc.).

Is summary, for our experiment we are looking for a model described by hypothesis

$$h(\boldsymbol{x}) \approx \begin{cases} 1 & \text{if } \mathsf{ORSP}(s_{\boldsymbol{x}}) = \text{“malicious”}, \\ 0 & \text{otherwise.} \end{cases}$$

For our testing, we took one week of data, from May 6$^{\text{th}}$ to May 12$^{\text{th}}$, 2019, and randomly sampled 1.5 million records from both the VirusTotal file and URL feeds. These records were already deduplicated using our deduplication cache (cf. Section 5.1), yet we further removed any records referring to the same file or URL. Then we queried ORSP for verdicts for all the samples and only kept those records for which ORSP had a known verdict. The collection of ORSP verdicts was performed at various times [3] during the end of May 2019, when all the collected records from VirusTotal were at least 10 days old, to give our backend systems time to process more samples in the meantime and thus ORSP providing us more verdicts. Due to this delay, we did not take the time difference between the individual records in the dataset into account.

This resulting dataset was the base for our experiments. We split it with ratio 70:30 into a training and testing dataset to train and test our model. Table 5.2 shows the number of records that we collected in the described process and Figure 5.2 shows the cumulative frequency of records, which had at least d positive detections.

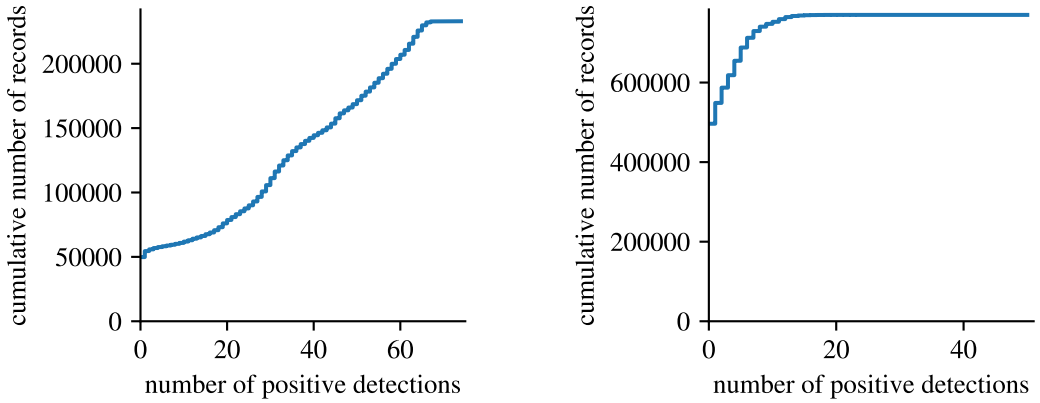
Then we used Apache Spark to train a binary classifier based on the logistic regression algorithm. We trained the model using the elastic net regularization with hyperparameters of $\alpha = 0.1$ and $\lambda = 0.1$ (tuned by grid search optimization). The code used to train the model is shown in Listing 5.2. For details about the LogisticRegression model in Spark, please refer to the documentation [15].

[3] Due to throughput limitations of the ORSP service available to us, we were not able to collect all the verdicts at the same time. Instead, the collection was performed over the course of multiple days.

Table 5.2: Number of records used for the experiment on VirusTotal feeds.

Feed	Total	Sample	Without duplicates	ORSP verdict	ORSP positive	Training	Testing
File	9,208.2	1,500	1,427.5	233.0	146.4 (63%)	163.4	69.6
URL	19,525.0	1,500	1,381.0	770.0	163.1 (21%)	539.5	230.5

The numbers of records (in thousands) are based on data we collected and processed from the VirusTotal file and URL feeds in the week from May 12$^{\text{th}}$, 2019. The columns (in order) describe the total number of records received, size of the random sample subset, number of records with unique samples (SHA-1 hash or URL) in the sample, number of those records for which ORSP has a known verdict, number and ratio of records with positive verdict by ORSP, and finally the sizes of the training and testing datasets.



(a) VirusTotal file feed (b) VirusTotal URL feed

Figure 5.2: Cumulative histogram visualizing the number of records with at least d detections in our experiment sample of VirusTotal data collected during the week from May 12$^{\text{th}}$, 2019.

Code Listing 5.2: PySpark code used to create and train (fit) the logistic regression classifier model.

```
classifier = LogisticRegression(
    regParam=0.1,
    elasticNetParam=0.1,
    maxIter=50
)
model = classifier.fit(training_data)
```

The logistic regression computes the probability p that a sample $s_{\boldsymbol{x}}$ with feature vector $\boldsymbol{x}$ would be declared as malicious by ORSP:

$$p := P(\mathsf{ORSP}(s_{\boldsymbol{x}}) = \text{“malicious”} \mid \boldsymbol{x})$$

The classifier f_t then predicts 1 if the probability is above certain threshold t and

0 otherwise.

$$f_t(\boldsymbol{x}) = \begin{cases} 1 & \text{if } P(\mathsf{ORSP}(s_{\boldsymbol{x}}) = \text{“malicious”} \mid \boldsymbol{x}) \geq t, \\ 0 & \text{otherwise,} \end{cases}$$

For comparison, we also created a simple model based only on a static rule, which predicts 1 if there were at least d positive detections, and 0 if there were less. We can describe it by a function

$$g_d(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i \geq d, \\ 0 & \text{otherwise,} \end{cases}$$

where $x_i = 1$ if the sample with features $\boldsymbol{x}$ was detected by the i-th vendor, and $x_i = 0$ otherwise.

For the predictor $f_t(\cdot)$ to be useful and worth the additional costs of model training, it must perform better than the simple $g_d(\cdot)$ function. We present and evaluate the results of our experiment in Section 7.1.8, where we also compare the performance of f_t and g_d.

CHAPTER 6

Security threats analysis

The security of the system has a very high priority. As we discussed in Section 4.5, the results of this system are used by other backend systems and researchers, therefore we need to have complete trust in the integrity of the data, which needs to be assured both by technical and by organizational means. In this chapter, we will discuss the security aspects of this system and summarize the outcome of our threat modeling.

When talking about the integrity of the data, it is important to understand that it begins already at the feed provider. This was highlighted by a security breach which happened at the MUTE feed endpoint in October 2018. Due to a vulnerability in a database administration tool Adminer [37, 22], the attacker was able to gain access to the local file system and database, including members' email addresses and hashed passwords. Although there is no indication that the published feed data was compromised, it shows that we need to take into account the possibility of a security breach outside of this system and even outside of the company.

6.1 Who is the adversary and what are the assets?

Before we discuss the threats in detail, it is useful to think about who the adversaries and their goals are. As a cyber security company, F-Secure is fighting against various kinds of electronic malicious behavior, with the aim to protect its customers. Any malicious actors could have an ambition to negatively affect the protection capabilities of F-Secure or damage its reputation. If they would succeed in compromising the Feed Automation system, they could observe or modify the data which the backend systems use for their processing or researchers for their analyses.

Therefore, these are the theoretical adversaries, who might benefit from attacking this system.

- Any actor who creates malware, phishing websites, etc. has an interest in

making antivirus or browsing protection less effective. This includes a wide range of possible attackers, from **vandals** causing damage to **criminals** seeking financial gain to **government agencies**. Their goal might be for example to block information which identifies their malware as malicious from reaching the company backends, or modifying the data to mark their phishing website as clean.

- Other **security companies** have an interest to make their protection work better than ours. Some company might want to improve their business by damaging our protection capabilities. Or they want access to our collected data in the data lake.
- A **disgruntled employee**, current or former, might want to hurt the company for personal reasons. In contrast to other attackers, employees can have inside knowledge of the system, and current employees might even have access to the credentials or the system and data itself.

The following are the key assets which we need to protect.

- The backend systems.
- The data lake.
- The data from feeds.
- The credentials for API authentication.

6.2 Threat modeling

To identify potential security risks and evaluate the security measures, we performed a threat modeling session together with the team maintaining this system, using the STRIDE methodology (cf. Section 2.7). In this section, we summarize some of the interesting findings which were identified.

6.2.1 Assumptions

During the threat modeling we make several assumptions. The project is deployed in AWS in an isolated network. We need to assume that Amazon enforces the security of the infrastructure. That includes the following:

- secure communication between all services (all connections between AWS services are encrypted with TLS),
- no unauthorized users or services can gain access to our services or data,
- nobody can tamper with the logs and metrics, etc.

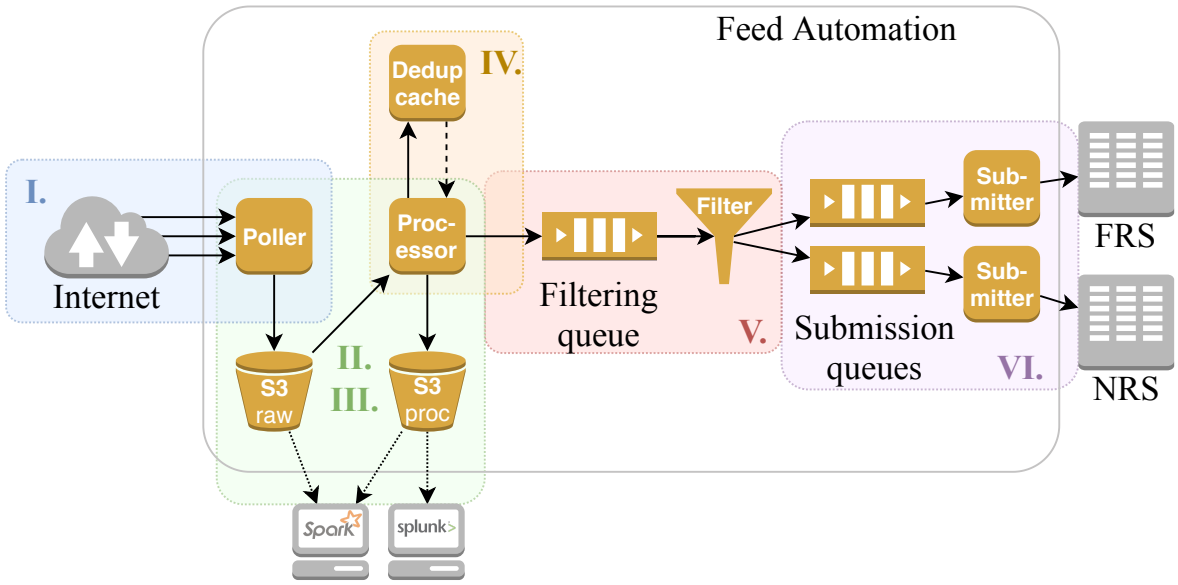


Figure 6.1: A simplified data-flow diagram between the components of the Feed Automation system.

6.2.2 Identification of security risks

We split up the data flow of Feed Automation into 6 main groups, as visualized in Figure 6.1, and we identified possible security risks of the system. In this section we present some of the findings.

I. Poller downloading from feed endpoint

- An attacker takes over the feed endpoint, allowing him to act as the provider and doctor the data. (**S**poofing, **T**ampering)
  Note: We use TLS connection to verify the provider's identity but we cannot detect breach of the provider's system. Once such a breach is discovered, we need thorough information about when and where did each record come from to revert any influence it might have had on our verdicts.
  Mitigation: Be in contact with the provider, turn off the feed intake if provider was compromised. Log more information about each downloaded package, including e.g. IP address of the endpoint.

- An attacker steals the TLS private key and is able to eavesdrop on the traffic or perform man-in-the-middle attack and alter the traffic. (**S**poofing, **T**ampering, **I**nformation disclosure)
  Mitigation: As above, be in contact with the provider, turn off the feed intake if provider was compromised.

- An attacker is able to make an endpoint send huge amounts of data, overloading our systems. (**D**enial of service)

Mitigation: Monitor the metrics for unexpected amounts of data coming from an endpoint, add alerts.

- An attacker is able to make an endpoint send corrupt data, breaking our proxies or poller. (**D**enial of service)
  Mitigation: Data is sanitized before processing. Monitor alarms for unexpected errors in processing.

- An attacker performs a denial-of-service attack on a feed endpoint, stopping flow to our services. (**D**enial of service)
  Mitigation: Monitor metrics and alarms for warnings of insufficient data. Contact the provider if the outage appears to be a serious problem.

II+III. Poller saving data, processor processing data

- An attacker can make the poller write to a different destination. (**T**ampering)
  Mitigation: Poller Lambda only has permissions to write to the S3 bucket with raw data.
  Recommended action: limit the permission only to the S3 prefix of corresponding feed.

- An attacker can make the processor write to a different destination. (**T**ampering)
  Mitigation: Processor Lambda only has permissions to write to the S3 bucket with processed data.
  Recommended action: Limit the permission only to the S3 prefix of corresponding feed.

- An attacker can make processor write sensitive data to data lake, where it is readable by a wider audience. (**I**nformation disclosure)
  Mitigation: The processor does not have access to any sensitive data, such as API credentials (only pollers can read and decrypt them).

- An attacker can overload the deduplication cache by sending too many records. (**D**enial of service)
  Mitigation: Not required. This would already have effect in earlier stages of the pipeline.
  Recommended action: Verify that the system flow does not stop if the deduplication cache fails (for whichever reasons) and stops responding.

- An attacker can send a faulty record which breaks other services (such as Splunk) which read the data from data lake. (**D**enial of service)
  Mitigation: All known problematic input is being sanitized. Monitor for unexpected parsing errors in other services which consume our data.

IV. Processor querying the deduplication cache

- An attacker is able to gain access to Redis from anywhere in the network. (**I**nformation disclosure,**T**ampering)
  Note: ElastiCache for Redis currently does not support any AWS-based authentication mechanisms.
  Mitigation: Evaluate the SecurityGroup used by the Redis caches and restrict access only to services which need the access (currently only the processors).

V. Filtering

- A filtering rule dropped a record without us knowing about it. (**R**epudiation)
  Mitigation: Improve logging.
  Recommended action: Add more detailed logging of dropped records and the corresponding rule which rejected it.

VI. Submitters

- Certificate expiration causes the submissions to halt. (**D**enial of service)
  Note: This is not directly a security threat but for example an expired FRS certificate could block all file submissions.
  Mitigation: Establish a procedure to regularly update the certificates.

CHAPTER 7

Evaluation of our solution

In this chapter we evaluate our deployed solution and reflect on the goals and requirements we defined in Chapter 3. We present both the measurable improvements introduced by this system, as well as discuss which requirements defined in Section 3.2 were successfully achieved and which still need to be addressed in future work.

7.1 Evaluation of the requirements

7.1.1 Centralization

The system was successfully deployed to production. Compared to the previous situation as we described it in Section 2.5.3, the system functionality is now located centrally in one place. Some feeds, which were previously downloaded by specialized single-task components in different accounts, in on-premise servers, or even at personal computers and maintained by various people around the company, were already successfully ported to the Feed Automation project. Thanks to this move, these similar tasks are now maintained by just a single team with clear responsibilities for the project, which frees up the earlier maintainers (often researchers or analysts) to spend more time working with the delivered data instead of the development and maintenance of specialized services for feed collection.

There are still some legacy feed downloaders remaining which have yet to be moved to this project. Those feeds which we have already moved to Feed Automation were appreciated by the previous maintainers and users of the data, for multiple reasons.

- The time-consuming development and maintenance was moved from them to a dedicated team.
- The additional functionality like efficient deduplication or centralized storage in data lake, which allows easy use of big data tools or Splunk, provides extra value which was not available earlier.

- The dedicated team maintaining this new project is quick to respond to bug reports or feature requests.

Summary: Requirement was satisfied for all the feeds which were already ported to the project. To fully complete this requirement, the remaining feeds need to be added to the project.

7.1.2 Cloud deployment

The system has been deployed and is running completely in the Amazon Web Services cloud. It has been designed utilizing the microservices paradigm based mostly on serverless functions, except where this was not possible due to technical limits.

Summary: Requirement satisfied.

7.1.3 Automation

The whole system is running autonomously. Feed endpoints are contacted in predefined intervals and data downloaded whenever available. Further services are started using triggers or queues. Thanks to the built-in retry behavior of the AWS Lambda functions, combined with the SQS messaging queues, we have achieved good fault tolerance, which has proven itself in many cases since the deployment of this system. For example, when our Internet proxies were temporarily down, the system regularly continued attempts until a connection was established, then downloaded all missing data and resumed normal operation.

The development and deployment to the cloud is following the DevOps practices of continuous integration and continuous deployment. Therefore also the testing and deployment is automated using Jenkins[1], which deploys the project from the company's internal Git.

Summary: Requirement satisfied.

7.1.4 Monitoring

The system provides many metrics, which can be observed either through AWS console or with Grafana dashboards. The development team is automatically alerted of any serious problems (such as insufficient data received from a feed) or failed deployments in Jenkins.

Summary: Requirement satisfied.

[1] Jenkins is an open source automation server. https://jenkins.io/

7.1.5 Costs

Deployment to Amazon cloud also enabled us to better monitor the costs of the system. Thanks to services like the *AWS Cost Explorer*[2] we are able to view the total costs but also explore the costs in more detail, such as per-service. This provides much better visibility than previously, when the costs were spread around the company and often hidden or shared with other services.

Due to the hidden costs of the previous solutions, we were unable to compare the total costs before this project and after. We were also unable to receive information from the backend systems about the financial impact that the new system is making thanks to reduced load. Secondly, as the current project includes feeds which were previously untapped and provides some new functionality, a direct comparison to the previous situation is not easily possible.

Summary: We have succeeded to make the overall costs easily observable. However, we were unable to perform a comparison of the running costs before this project and with the Feed Automation system.

7.1.6 Security

The security of the system was explored in detail during threat modeling in Chapter 6.

7.1.7 Easy management of feeds

Our aim was to make the additions, modifications, or removals of feeds in Feed Automation as simple as possible. To this end, we contained all feed-specific functionality and configuration just to 2 services, the pollers and processors. We provide some common functionality (such as typical API clients or tools for generating files for data lake) ready for use in the form of prepared classes, which only need to be subclassed. For example, to add a feed, a developer generally only needs to extend two classes and configure the corresponding Lambda functions (schedule, memory, etc.). After submitting these changes to the common git repository and receiving approval, the changes are automatically deployed.

To evaluate this requirement practically, we have observed and supported a fellow data engineer in adding a new feed processor. Although he had only minimal previous knowledge of the system, he considered it to be a simple and straightforward process. Nonetheless, we have identified some areas which could be further streamlined to simplify the feed-specific code by moving the complexity to the common superclasses.

Summary: To objectively evaluate this requirement, we would need to perform a thorough usability study with multiple engineers to see if we succeeded to make the management of feeds simpler and to identify areas which could further improve this aspect. Unfortunately, this was not possible during the given time frame of this project.

[2] https://aws.amazon.com/aws-cost-management/aws-cost-explorer/

Our goal was to make the process of adding a new feed so simple that it takes at most 2 days of developer's time. As the currently included feeds were added during the development of the whole system, we will only be able to properly evaluate this goal once we start adding more feeds during the upcoming work.

7.1.8 Reduction of data noise

Removal of irrelevant fields

Requirement achieved by the processors, which allow performing arbitrary changes on every processed record.

Deduplication

Removing duplicate records from the received data proved to be a very powerful tool for reducing the data volume without losing any important information. As we defined in Section 3.2, we consider a record an irrelevant duplicate if it provides us no new information about a sample. As a consequence of this definition, we look for duplicates only among earlier records from the same feed, not across all feeds. If we see a record about a sample X in feed A and later in feed B, that is a useful information to our analysts and we want to keep both records.

To evaluate the effectiveness of such deduplication, we present a few examples of how the ratio of duplicates per feed evolves when starting from *zero knowledge*, i.e. when the deduplication is empty at the beginning, over the course of the same 10 days. For these experiments, we used a dedicated cache, which we used exclusively for deduplication of records from a single feed.

The presented examples show that the effectiveness of deduplication depends strongly on the source feed. The average ratio of duplicates ranges from 0% to 99% between the feeds. The reason for such a difference between the feeds is generally how the data is published. Some providers, such as PhishTank, publish a regularly updated package, which contains largely the same information between two consecutive updates, only with minor changes – that results in a very high duplicate rate. Some other providers, such as MUTE, only publish information about new samples which have not yet previously appeared in the feed – that results in a low or zero duplicate rate. In between these two extremes are feeds which publish new information as it appears, whenever there is new information about a sample. Good examples are the VirusTotal feeds, where the rate of duplicates varies.

In the following figures, for each example feed we show three graphs which show various metrics during the same 10-day period. The top graph shows how many records we processed in total and how many were accepted as not duplicates, i.e. either a new record for a sample we haven't seen before, or a "modified" record for a sample we have seen earlier with different metadata. The middle graph shows the percentage of "new", "seen", and "modified" records at different points in time, with the thin lines showing the actual values and the thicker lines representing a running

average of the surrounding ±12 hours. The bottom graphs shows how much memory was used by the deduplication cache to store the information. We did not include the bottom graph for PhishTank and MUTE feeds, as the storage needs are minimal.

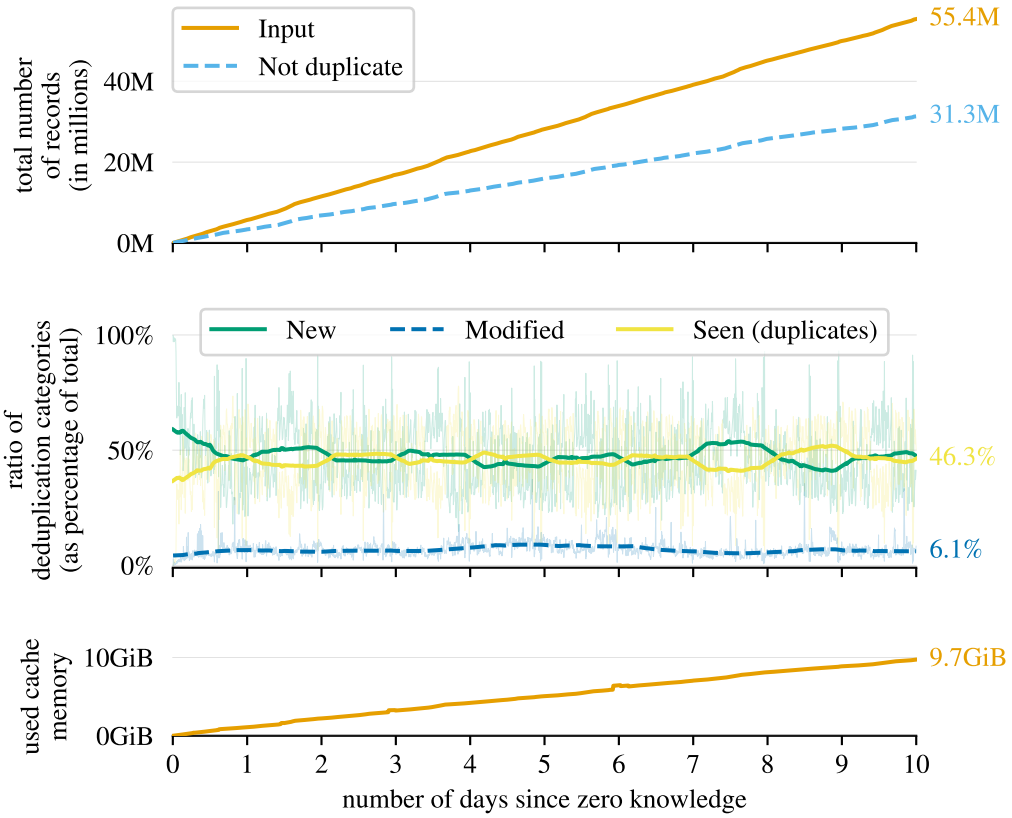


Figure 7.1: The effect of deduplication on VirusTotal URL feed, when starting with an empty dedicated deduplication cache.

The **VirusTotal** feeds are currently the most frequently updated feeds which we process. In Figure 7.1 we show the effect of deduplication for the VirusTotal URL feed. We can see that the ratio of detected duplicates (yellow line) goes up vey quickly and stays around 46% on average. Secondly, we can also see the ratio of "modified" records (blue line). This shows that thanks to the "smart" deduplication, we were able to detect on average 6.8% of records which we have seen already before but which contain important new information. If we had used just naive deduplication, based only on the URL, these records would have been discarded as duplicates. The bottom graph tells us that the information about VirusTotal URL feed records uses almost 1 GB of storage per day.

Another interesting high-volume feed to look at is the **Microsoft Bing Malicious URLs** feed, which is shown in Figure 7.2. Unlike VirusTotal feeds, which are updated regularly every minute, this Microsoft feed is updated in big batches at irregular intervals, therefore it shows steps. The interesting thing to point out is that it takes longer for the deduplication to reach its maximum effectiveness, which can

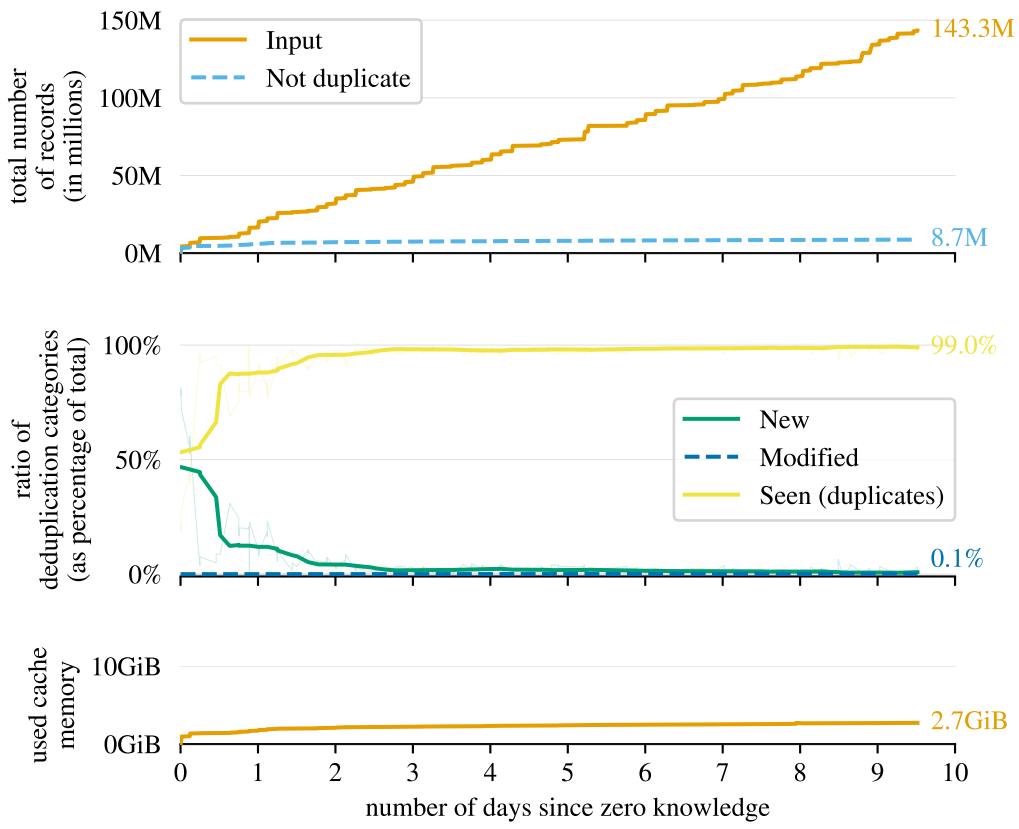


Figure 7.2: The effect of deduplication on Microsoft Bing Malicious URLs feed, when starting with an empty dedicated deduplication cache.

be up to 99% on average. This means that a short deduplication window of e.g. one day would be significantly less efficient than a long one. We can also see that due to the high ratio of duplicates, the storage needed for this feed rises only minimally after a few days.

Next we will show two examples with completely opposite ratios of duplicates. In Figure 7.3 we show the **PhishTank** feed and in Figure 7.4 the feed from the **MUTE Group**. As we explained in Section 2.2, the PhishTank feed is published as an hourly updated package, so a lot of the records will remain unchanged between subsequent packages. This leads to a very high percentage of duplicates in this example. After deduplicating, we get a much smaller flow consisting only of the new or changed phishing sites.

In contrast, the MUTE Group's feed of malicious URLs appears already cleaned at the source, and we detect no duplicates at all.

Rule-based filtering

The rule engine we introduced in Section 5.2.1 allows for rules which are easily configurable even by users with limited programming knowledge. Given more time,

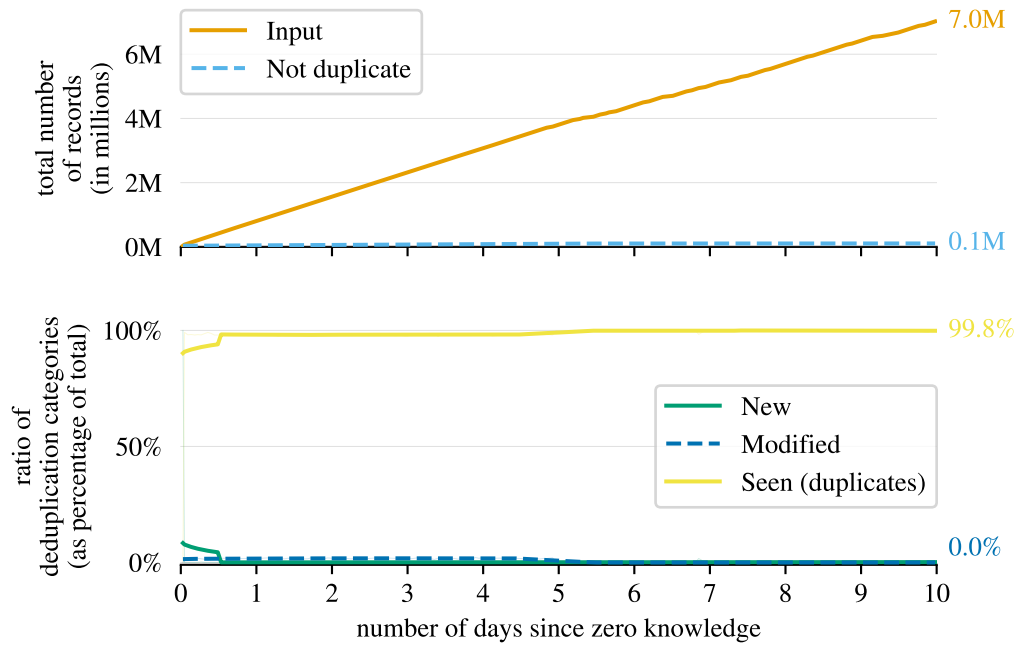


Figure 7.3: The effect of deduplication on the PhishTank feed, when starting with an empty dedicated deduplication cache.

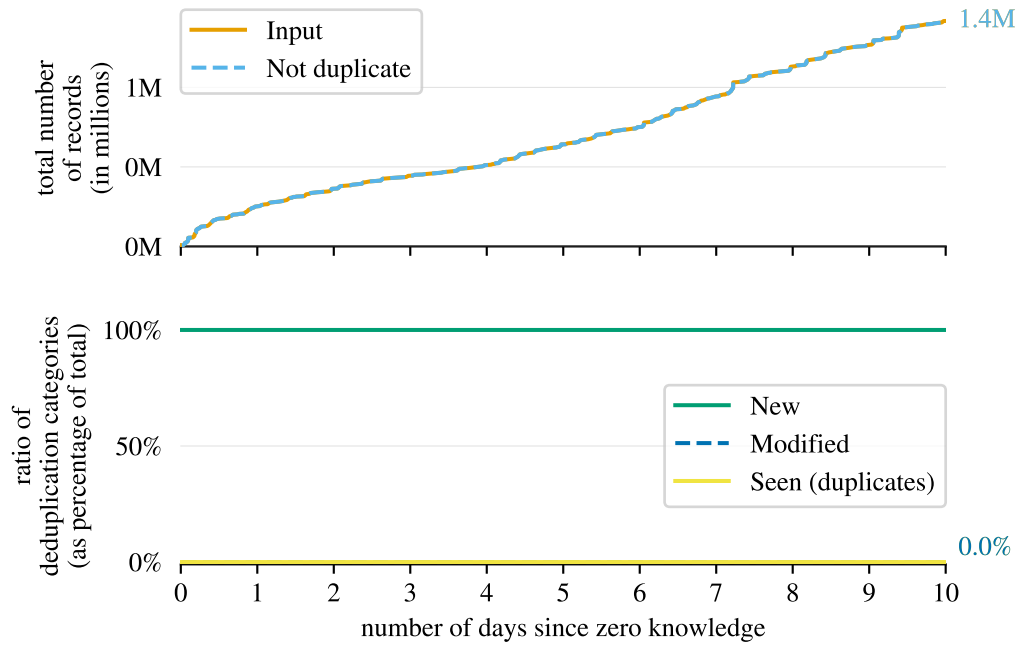


Figure 7.4: The effect of deduplication on the MUTE feed, when starting with an empty dedicated deduplication cache.

it would be beneficial to perform a thorough user study to properly evaluate the

usability of these rules for the researchers and analysts at the company.

The effectiveness of the rule engine to reduce the amount of data naturally depends on the rules themselves. In most feeds, there is no reason to filter out any records at all. But as an example, the rule given in Listing 5.1, which drops any records with zero positive detections and no categories, was able to discard almost 30% of records, as Table 7.1 shows.

Table 7.1: Percentage of records accepted by the filter for the VirusTotal URL feed during the first 5 months of year 2019.

January	February	March	April	May	Total average
68.3%	71.1%	71.7%	64.9%	77.4%	70.6%

Dynamic filtering

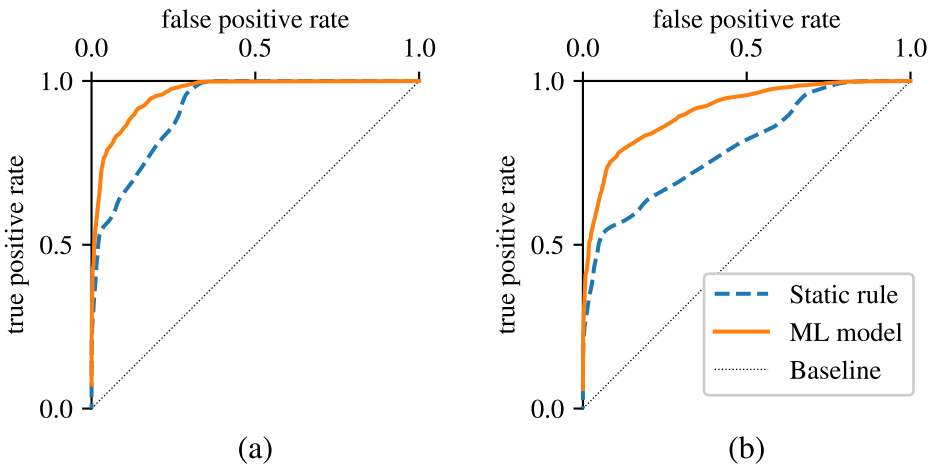
To evaluate the feasibility of the dynamic filtering approach, we compared the performance of a classifier based on a logistic regression model with the performance of a simple static rule-based approach which only takes into account how many positive detections were reported (not by which vendor).

Each of these classifiers can be configured to accept more or less records (i.e. predict value 1) by adapting the threshold, i.e. the minimal probability for the logistic regression or the minimal number of positives in the rule. When configuring this value, we can find the balance between the number of false positives and true positives. The setting depends on the needs (e.g. how much data the backend systems can currently accept) and there is no one correct value.

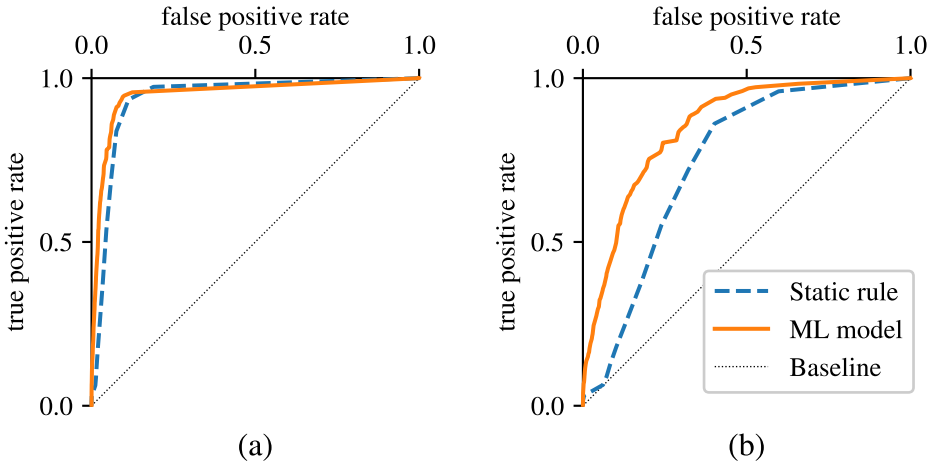
Therefore, to compare the two approaches, we use the receiver operating characteristic (ROC) curve, which plots the true positive rate (TPR) against the false positive rate (FPR) as the threshold is varied. This allows us to compare the performance of the classifier over the whole range of possible thresholds. To numerically compare the overall performance of a classifier, we use the "area under curve" (AUC) metric, which measures the area below the ROC curve on the $[0, 1]$ range.

For each of the two feeds, we evaluated first on the complete testing data set, but additionally also on a subset of the testing dataset with only those records which have at least one positive detection. The records with zero detections will be trivially evaluated to 0 by any meaningful classifier (unless the threshold is set to 0, i.e. accepting all records), therefore it is interesting to see how the classifiers perform on the subset which contains only records with at least one detection. In the VirusTotal file feed, 47% of the records in our sample had at least one positive detection, and 31% of the records in the URL feed.

First, we calculated the area under the ROC curve (AUC) for each of the classifiers. We compared the values for both feeds and both for the complete testing dataset and for the subset only with records with at least one detection. This provided us



(i) Trained and evaluated on the **VirusTotal file feed**.



(ii) Trained and evaluated on the **VirusTotal URL feed**.

Figure 7.5: ROC curves comparing the performance of a classifier based on logistic regression (ML model) and a classifier based on static rule. (a) Evaluated on the complete testing dataset. (b) Evaluated on subset of the testing dataset only with records with at least one detection.

with 4 pairs of values, which are presented in Table 7.2. As we can see, the logistic regression ML model performs better in each case, although the difference for URL feed with complete testing test is small.

To better understand the performance of the classifiers, we further analyze the ROC curves. In Figure 7.5i we compare the curves for the file feed and we can see that the logistic regression model outperforms the static rule. Particularly in the right plot when applied only to records with detections (Figure 7.5i(b)), we can see that we get considerably lower rate of false positives for higher values of true positive rate.

Figure 7.5ii shows the ROC curves for the VirusTotal URL feed. Also here the

Table 7.2: The area under the ROC curves for the classifier based on logistic regression (ML model) and classifier based on static rule, when evaluated on the complete testing dataset and on a subset of only records with at least one detection.

	Complete testing dataset		Testing subset, only with detections	
Feed	ML model	Static rule	ML model	Static rule
VirusTotal file feed	0.961	0.915	0.909	0.799
VirusTotal URL feed	0.951	0.937	0.852	0.745

logistic regression model achieves better results when applied to the dataset with positive detections (Figure 7.5ii(b)) but does not show significant differences when applied on the complete dataset (Figure 7.5ii(a)).

Summary: Overall, the classifier based on the logistic regression ML model shows better performance than the static rule and good promise for a possible future application.

However, for our current production system, it was decided there is not a need for this type of filtering at this moment, because we only drop certain records with zero positive detections and there is no advanced model required for this. In the future, if the amount of data would increase significantly, we will reconsider whether to apply this advanced filtering technique. To do that, we would perform a detailed analysis of the costs, particularly the costs of collecting the labeled data and training the model, to see whether the reduction of data volume by the advanced filtering outweighs its costs and added complexity.

It would be beneficial to perform more thorough tests. In particular, we would like to analyze different data samples and see how the delay between training of a model and the time of its evaluation influences the quality of the predictions. Unfortunately, performing more experiments is currently hindered by the speed at which we are able to collect ORSP results – the backend service we use was not designed to perform millions of sample queries in short time, as we need for our evaluation. We hope this limitation is only temporary and will be improved in the next versions of the backend service.

Summary of data noise reduction

Our minimal goal was to achieve a reduction in the number of records, compared to the records coming in from the feeds, while setting an additional ambitious target of reaching at least 50% reduction. In Table 7.3 we show the statistics for one example week and show that we achieve ca. 70% overall reduction, when comparing the number of records of records coming in from the feeds and the number of records we submit to the backend. The reduction is achieved mainly on the URL feeds with an average reduction of 78.4%, while the file feeds are reduced only by 11.4% (mainly

due to low rate of duplicates in the file feeds currently imported and processed by Feed Automation).

Table 7.3: The number of records (in thousands) coming in from the feeds and submitted out of the Feed Automation system to the backend, after processing, deduplication and filtering. Based on values over the course of one week (May 6th–12th 2019).

Feed	Input	Output	Ratio
AV-TEST Windows	279.4	279.4	100.0%
VirusTotal file feed	10,254.6	9,046.3	88.2%
VirusTotal Hunting not.	31.8	31.3	98.5%
File feeds total	10,565.8	9,357.0	88.6%
Bing Malicious URLs	32,569.7	604.7	1.9%
MUTE Group	354.6	354.6	100.0%
OpenPhish	6.9	6.9	100.0%
PhishTank	2,039.9	48.7	2.4%
VirusTotal URL feed	37,336.8	14,591.3	39.1%
URL feeds total	72,307.9	15,606.2	21.6%
All feeds total	82,873.7	24,963.2	30.1%

Summary: We have achieved our overall target of reducing the number of records by more than 50% but only thanks to the URL feeds, while the average for file feeds is well below this goal.

7.1.9 Output

First, all the data received from the feeds is saved "raw" (with minimal alterations) to the data lake. Secondly, the processed and deduplicated data generated by the processors is also saved to the data lake. The processed, deduplicated, and filtered records are automatically submitted to the backend systems, the File Reputation System and the Network Reputation System, to provide new information to the Security Cloud knowledge base.

Measuring the exact time it takes for individual records to go through the whole system is not easily directly measurable but we can estimate it from the running times of the individual services. We calculated the average hourly running times for each service over the course of one week. Out of these values, we compute the statistics shown in Table 7.4. Note that the significantly greater maximum for file submitter is caused by occasional download and upload of very big files.

We can see that the total average processing time from polling form published to submitting to backend is only about 10.4 and 12.5 seconds for URLs and files, respectively. To estimate the total time of the whole flow, we also need to include the delay caused by communication between the services, mainly the waiting time in

Table 7.4: Statistics of hourly average running times of individual services per package or SQS message. Based on values over the course of one week (May 6th–12th 2019). All times are in seconds.

		Hourly average running time		
Service	Unit of work	Minimum	Average	Maximum
Pollers	Package	0.5	5.8	74.2
Processors	Package	1.4	4.3	96.1
Filter	Message	0.2	0.2	0.3
URL submitter	Message	0.1	0.1	0.2
File submitter	Message	0.1	2.2	2040.0

queues. For the filtering and URL submission queue, the average age of oldest message in queue (i.e. the longest waiting message) was 3.5 and 8.2 seconds, respectively. For the file submission queue, this value was unfortunately not usable[3].

Summed up, we estimate that the complete flow of URL records takes on average approximately 22.1 seconds. For file records, we can currently only estimate the time between polling of the data until it reaches the submission queue, which is on average 16.0 seconds.

Summary: Requirement to save records to data lake and submit to backend is satisfied. The estimated average time for the complete flow of data from polling to submission is well below the goal of 5 minutes. However, the estimates are not exact and we are missing values for the waiting time in the file submission queue. During future work we would like to add a more precise timing method, for example by passing a timestamp of polling with each message.

7.1.10 Complete searchability and quick lookup

All the collected data, along with other datasets in the company's data lake, can be analyzed, searched, and processed using big data tools provided for the users, such as Apache Spark. Additionally, we index the processed data using Splunk, which provides a simpler user interface for any authorized user to search in this data and perform advanced queries combining multiple indexes – not just from Feed Automation but also other sources and tools.

The deduplication cache keeps a useful overview of all the samples we have recently encountered in any of the feeds and when. We have prepared the functionality for

[3]The reason why we could not use this metric to estimate the average waiting time was that the SQS metric ApproximateAgeOfOldestMessage also includes messages which encountered an error during the submission. For file records, this often happens when the sample cannot be downloaded from the Internet. If a record could not be submitted, the submission will be postponed until a later time by returning it to the queue and hiding it for an iteratively increasing period of time. This strategy increases fault tolerance but causes the metric to show higher times than the actual average waiting times of messages in the queue.

providing this information to outside users and systems by the means of a REST API. It is not yet completely ready to be deployed, as it still requires more detailed configuration of the access control, so that only authorized systems and users can query this API.

Summary: The searchability requirement is satisfied by providing our users, such as malware researchers, with powerful ways of searching over this data. The requirement of quick lookup is currently not satisfied, as we were not yet able to deploy the API providing a fast lookup ability over the samples encountered recently in the feeds.

CHAPTER 8

Related work

The difficulties of security-related intelligence are well known. In 2009, Camp et al. [17] described some of the challenges. They argue that even though sharing data, such as malware samples, URLs found in spam, or network data, is invaluable for cyber security research and for development of new defenses, it is often hard for researchers to get access to such data, as the service providers lack interest to share data with researchers, often due to legal or privacy concerns. They also describe how different types of data require different types of delivery mechanisms – as a continuous feed, periodically updated package, or a one-time archive. Finally, they propose a list of examples of data which would be useful for the needs of security researchers.

Dumitraş et al. [33, 32] from Symantec Research Labs also recognize the problem of insufficient availability and sharing of security-related data for experimental research. To tackle the issue, they propose a security-benchmarking framework, called the *Worldwide Intelligence Network Environment*. This framework makes a dataset of representative field data available to researchers worldwide, and additionally provides a platform for repeatable cyber security experiments with these data sets. Researchers can apply for access to this framework from Symantec.

There are also other data sharing platforms, for example:

- The *Security Information Exchange* (SIE) [36], originally by the Internet Systems Consortium and acquired in 2013 by Farsight Security, Inc. [35] provides traffic data using its passive DNS technology from many contributors worldwide.
- The *Protected Repository for the Defense of Infrastructure Against Cyber Threats* (PREDICT) was an initiative by the U.S. Department of Homeland Security, which aimed to form a "partnership between government, critical information infrastructure providers, and the security development community (both academic and commercial)" and provide "researchers, developers, and evaluators with regularly updated network operations data sources relevant to

cyber defense technology development." [69] In 2015, the PREDICT project was superseded by the *Information Marketplace for Policy and Analysis of Cyber-risk & Trust* (IMPACT) project [68, 43].

- The *Internet Measurement Data Catalog* (IMDC) [18] was developed by the Center for Applied Internet Data Analysis at the University of California to "provide a searchable index of available (Internet) data, enhance documentation of datasets via a public annotation system, and advance network science by promoting reproducible research" [63]. IMDC stopped in 2018, after the funding of the project had ended.
- Internet Traffic Archive (ITA) [25], sponsored by Association for Computing Machinery (ACM) SIGCOMM, is a pubic-domain repository of filtered traces of Internet network traffic.

The international defense alliance NATO recognized the strong need for automated sharing of quality-assured security-related data and tasked the NATO Communications and Information Agency to develop the *Cyber Security Data Exchange and Collaboration Infrastructure* (CDXI) knowledge management tool. Dandurand and Serrano [29] define that the objectives are to "(1) facilitate information sharing, (2) enable automation, and (3) facilitate the generation, refinement and vetting of data through burden-sharing collaboration or outsourcing." In their work, they discuss many of the challenges still present in the process of automated information sharing in the cyber security domain, such as different sources with inconsistent data, large volumes, various not interoperable protocols and access mechanisms, incompatible semantics, and varying quality of the data. They define 11 high-level requirements for the CDXI capability and propose a high-level architecture for the tool.

Nowadays, probably the most widely used [46, 61] standards for threat intelligence sharing are the *Structured Threat Information Expression* (STIX) language together with the *Trusted Automated Exchange of Indicator Information* (TAXII) protocol. These standards were initially started by the US Department of Homeland Security in 2012 and 3 years later licensed it to the Organization for the Advancement of Structured Information Standards (OASIS), a nonprofit consortium that drives the development, convergence, and adoption of open standards. STIX [23] is a language and serialization format used to describe and exchange intelligence about cyber threats. It uses a graph model to describe relationships between various Domain Objects to describe the cyber threat intelligence. A Domain Object can be one of Attack Pattern, Campaign, Course of Action, Identity, Indicator, Intrusion Set, Malware, Observed Data, Report, Threat Actor, Tool, or Vulnerability. The default serialization format of STIX Version 2 is JSON (used to be XML for version 1) but other formats can also be used. TAXII [24] is an application layer protocol used to exchange the intelligence about cyber threats over HTTPS, using a RESTful API. While not limited to STIX, it has been designed specifically for transferring cyber threat intelligence represented in the STIX language. While it seems that the use of STIX and TAXII is increasing, unfortunately none of the feeds that we have included in this work support it.

Multiple studies comparing the sharing standards and platforms have been published in recent years. Kampanakis [46] lists and describes the available security intelligence sharing options. Goodwin and Nicholas from Microsoft [40] created a general guide for cyber security threat exchange, discussing the challenges, models, and methods of exchange, data formats, etc. Finally, they give a set of recommendations for anyone developing a new intelligence sharing system. Sauerwein et al. [61] examined the state of the art of threat intelligence sharing platforms. Menges and Pernul [54] thoroughly compare and evaluate state-of-the-art incident reporting formats and advise readers about suitable use cases.

CHAPTER 9

Conclusion and future work

Efficient and fully automated collection and processing of data from very heterogeneous sources poses a challenge. Especially in the cyber security industry it is important to gather as much information as possible about malicious files, websites, and activities, in order to provide best possible protection to the users of security software. In particular, proactively gathering information from third parties is crucial to achieve protection for the "first customer", i.e. the first user of a security software who encounters a malicious file or visits a malicious website. Without getting information about these samples from a different source, the security provider can only wait and react *after* their first customer encountered the malware and possibly got infected, scammed, etc.

Having collected all this data, we encounter the other problem – how to separate the useful information from the data noise? Processing unnecessary data leads to unnecessary costs or might even be unfeasible if the amount becomes too large.

In this work, we presented the cloud-based **Feed Automation** system, which centralizes and fully automates the collection and processing of data from various sources, in varying formats, schemata, sizes, etc. It also provides powerful yet flexible options to reduce the noise in the data, both by the means of a smart deduplication process, which takes into consideration more than just the record identifier, and by static logic rules, easily configurable by the analysts. Furthermore, we propose ways to achieve additional dynamic filtering based on machine learning (ML) models, which could be used to remove records from some feeds based on the probability of their usefulness.

The whole system has been successfully deployed in the Amazon Web Services cloud and we show how such a data processing system can be designed as a collection of microservices based on serverless functions. This approach brings a number of benefits, such as the pay-as-you-go billing model of functions (not paying for idle time), easy maintenance, scalability, fault tolerance, and a great flexibility to extend the system. Thanks to the deployment in the Amazon cloud, we gained detailed

insight into the costs of the system. We can also take advantage of various readily provided services for security, logging, monitoring, alarms, etc.

During evaluation of the system, we have shown that we can achieve very significant reductions in the amount of data by efficient smart deduplication, without losing any, or only minimal, information, in contrast to filtering, where static rules determine which records we drop and what information we lose. Therefore we determine that efforts should always concentrate first on removing the duplicates and only if the loads are still too high even after deduplication, look for ways to filter the data with rules or ML models.

The described system has been deployed and is currently running in the production environment of F-Secure, already providing value to the backends and ultimately helping to better protect the customers. It has become one of the core components of the new-generation backend for analyzing samples and providing reputation service for files and websites.

Additionally, the deployment of this cloud-based system allows the disabling of other legacy downloaders, contributing to the big move of all F-Secure services from on-premise servers to Amazon Web Services.

We expect this project to be further developed and extended, to provide new functionality to the other services and users in the company.

The next key step will be to include more feeds into the project, with the closest goal of replacing all legacy downloaders and becoming the sole central system for receiving all feed-like data from other providers.

As we described in Section 4.2.4, the system already supports alternate data sources besides feeds, such as crawlers and spam. There have been requests for extending the functionality of these data sources. Particularly the spam component could provide a lot more value, but requires additional processing of the records, which the current system does not support yet. We would also like to explore other interesting data sources, such as crawlers of social media, e.g. Twitter.

An interesting new addition would be to import feeds of *indicators of compromise* (IoC). These records indicate possible security breaches, and are useful for the incident response teams, but also for the malware researchers and analysts. This would mean a new, incompatible record type. Because Feed Automation was initially built only to support records referring to a file or a webpage, including IoCs into the system would require substantial modifications, but it would provide a very valuable new source of information.

We want to explore the possibility of ingesting internal streams of data, not only external feeds. Some of these data streams provide very useful information. For example, a stream of logged requests to the ORSP system (querying about the "reputation" of a file or a webpage) would provide a valuable information about the prevalence of those samples. However, these streams have such high volumes that

there is currently no system deployed capable of handling all the data to generate new intelligence. This is where an efficient deduplication service like the one in Feed Automation could help to reduce the flow to a manageable size. Because this is an internal stream of data from the customers, we would – besides technical challenges – also have to address other new concerns, such as privacy of the requests from our customers.

There are also some interesting technical opportunities. Amazon continuously creates new services which we might take advantage of. Last year, Amazon launched a new high-performance graph database engine called *Amazon Neptune* [5], describing it as "optimized for storing billions of relationships and querying the graph with millisecond latencies". That opens up interesting possibilities of storing the data we collect about files and websites in a graph, which would make it faster to access, and open up new possibilities to traverse and understand the relationships between the different samples.

We intend to explore other formats for storing the data in Amazon S3. Currently, we store all the data in JSON Lines format, compressed with gzip (cf. Section 4.2.1). JSON has the great advantage of being widely supported and fairly easily readable by humans. However, it is not very efficient for storing big amounts of data. A promising alternative is the Apache Parquet[1]. Unlike JSON Lines, which is stored in file line-by-line, Parquet stores data physically column-by-column. This can save considerable space, because there is no need to save the column names for each record, the compression is more efficient if the column contains a lot of similar values, and the format also supports dictionary encoding to encode categorical values. Furthermore, reading only a subset of the columns from a file is faster. One of the main disadvantages is that due to the column format, the data must have a flat schema. For a schema that goes multiple levels deep (like VirusTotal), this requires restructuring the data. Furthermore, it is a risk to commit ourselves to a less-widely used format.

As the processing of each feed incurs costs to the company (license fees, AWS costs, development and maintenance costs), we would like to have information how useful each feed is and whether the value it brings to the company is worth the costs. Defining such metrics is a very challenging task, which requires good understanding of the information in the feed data, how it is used in the company by the backend systems, analysts, and researchers, and also requires feedback from the backend systems. One of the initial steps should be to observe the overlap between the feeds and the time when the records appear. For example, if 99% of the records from feed A are present also in feed B, we might be ingesting feed A needlessly. But if the records in feed A appear a day earlier than in B, the costs might be worth the earlier delivery.

Finally, this project only concentrated on receiving intelligence from other partners. A logical next step would be to explore the methods currently used to share

[1] https://parquet.apache.org/

the company's own threat intelligence to other partners, and possibly introduce a centralized system for publishing such information, with some of the standards described in Chapter 8, such as STIX and TAXII.

APPENDIX A

Feed Automation

A.1 Complete system architecture

In Figure A.1 we present the complete architecture of the Feed Automation system. The diagram includes other data sources, such as crawlers or spam feeds, which we have only briefly described. Although these components were initially not parts of this project, they utilize the system's functionality – the deduplication cache, the filter, and the submitters. The diagrams also shows some of the AWS services which were used to build the Feed Automation system.

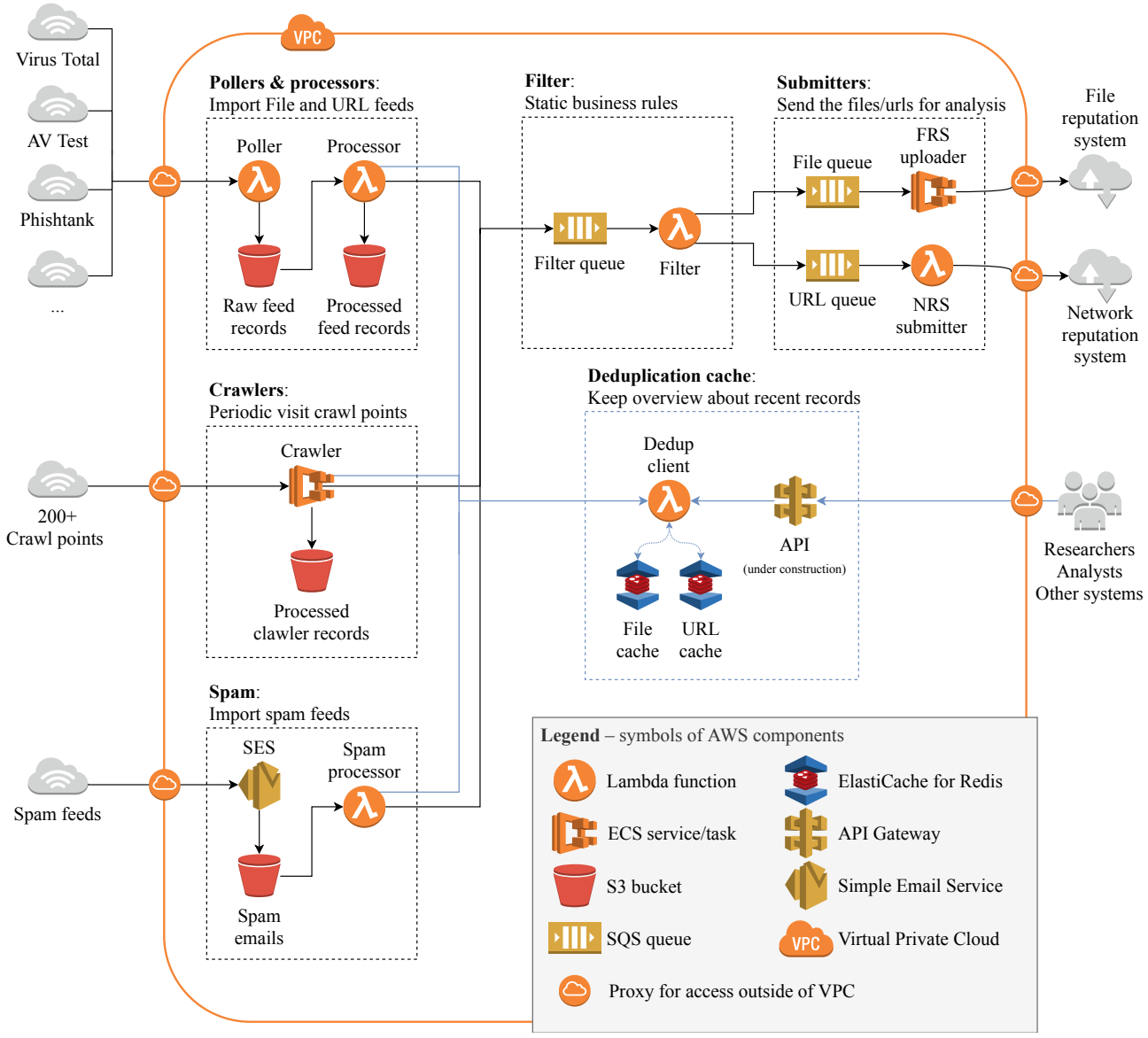


Figure A.1: Schema of the Feed Automation system.

APPENDIX B

Feeds

B.1 VirusTotal

Listing B.1 shows an example file report from VirusTotal for an EICAR test file [14]. Note that the reports may be much bigger, depending on the behavior of the file.

Code Listing B.1: VirusTotal file scan report for an EICAR test file, in JSON format. Some data was omitted to reduce size. Retrieved from on June 6th, 2019.


```
{
  "sha256": "131f95c51cc819465fa1797f6ccacf9d494aaaff46fa3eac73ae63ffbdfd8267",
  "sha1": "cf8bd9dfddff007f75adf4c2be48005cea317c62",
  "md5": "69630e4574ec6798239b091cda43dca0",
  "scan_id": "131f95c51cc819465fa1797f6ccacf9d494aaaff46fa3eac73ae63ffbdfd8267
     -1559348089",
  "permalink": "https://www.virustotal.com/file/131
     f95c51cc819465fa1797f6ccacf9d494aaaff46fa3eac73ae63ffbdfd8267/analysis
     /1559348089/",
  "scan_date": "2019-06-01 00:14:49",
  "first_seen": "2006-05-23 17:26:21",
  "last_seen": "2019-05-24 11:42:56",
  "times_submitted": 1716,
  "community_reputation": 213,
  "malicious_votes": 19,
  "harmless_votes": 71,
  "size": 69,
  "type": "Text",
  "positives": 58,
  "total": 64,
  "scans": {
    "Avast": {
      "detected": true,
      "result": "EICAR Test-NOT virus!!!",
      "update": "20190601",
      "version": "18.4.3895.0"
    },
    "Avira": {
```

```
      "detected": true,
      "result": "Eicar-Test-Signature",
      "update": "20190601",
      "version": "8.3.3.8"
    },
     ... skipped 60 scan results ...
    "Panda": {
        "detected": false,
        "version": "4.6.4.2",
        "result": null,
        "update": "20190531"
    },
    "Yandex": {
      "detected": true,
      "result": "EICAR_test_file",
      "update": "20190531",
      "version": "5.5.2.24"
    },
    "Zoner": {
      "detected": true,
      "result": "EICAR.Test.File-NoVirus.250",
      "update": "20190531",
      "version": "1.0"
    }
  },
  "vhash": null,
  "authentihash": null,
  "unique_sources": 936,
  "ssdeep": "3:a+JraNvsgzsVqSwHqN:tJuOgzsky",
  "resource": "cf8bd9dfddff007f75adf4c2be48005cea317c62",
  "response_code": 1,
  "additional_info": {
    "magic": "ASCII text",
    "sigcheck": {},
    "exiftool": {
      "FileAccessDate": "2015:02:17 17:23:18+01:00",
      "FileCreateDate": "2015:02:17 17:23:18+01:00"
    },
    "trid": "EICAR antivirus test file (100.0%)",
    "positives_delta": 0,
    "autostart": [... skipped ... ],
     ... skipped more entries about sample parents ...
  },
  "tags": ["text", "attachment", "via-tor"],
  "submission_names": [
    "eicar.com", "malicious.txt", "abc.txt",
     ... skipped further 97 submission names...
  ],
  "ITW_urls": [
     ... skipped 100 ‘‘in the wild’’ urls ...
  ],
  "verbose_msg": "Scan finished, information embedded"
}
```

APPENDIX C

Amazon Web Services

C.1 ElastiCache node types and pricing

Tables C.1 and C.2 list the currently available ElastiCache node types and their corresponding on-demand pricing, as per April 2019 [4].

Table C.1: Available standard ElastiCache node types of current generation, and their on-demand pricing (April 2019).

Cache Node Type	vCPU	Memory (in gibibytes)	Network Performance	Price Per Hour (in US dollars)
cache.t2.micro	1	0.6	Low to Moderate	0.432
cache.t2.small	1	1.6	Low to Moderate	0.864
cache.t2.medium	2	3.2	Low to Moderate	1.752
cache.m4.large	2	6.4	Moderate	4.128
cache.m4.xlarge	4	14.3	High	8.232
cache.m4.2xlarge	8	29.7	High	16.464
cache.m4.4xlarge	16	60.8	High	32.952
cache.m4.10xlarge	40	154.6	10 Gigabit	82.392
cache.m5.large	2	6.4	High	4.128
cache.m5.xlarge	4	12.9	High	8.232
cache.m5.2xlarge	8	26.0	High	16.464
cache.m5.4xlarge	16	52.3	High	32.952
cache.m5.12xlarge	48	157.1	10 Gigabit	99.072
cache.m5.24xlarge	96	314.3	25 Gigabit	198.144

Table C.2: Available memory optimized ElastiCache node types of current generation, and their on-demand pricing (April 2019).

Cache Node Type	vCPU	Memory (in gibibytes)	Network Performance	Price Per Hour (in US dollars)
cache.r4.large	2	12.3	Up to 10 Gigabit	0.254
cache.r4.xlarge	4	25.1	Up to 10 Gigabit	0.507
cache.r4.2xlarge	8	50.5	Up to 10 Gigabit	1.014
cache.r4.4xlarge	16	101.4	Up to 10 Gigabit	2.028
cache.r4.8xlarge	32	203.3	10 Gigabit	4.056
cache.r4.16xlarge	64	407.0	25 Gigabit	8.112
cache.r5.large	2	13.1	Up to 10 Gigabit	0.241
cache.r5.xlarge	4	26.3	Up to 10 Gigabit	0.480
cache.r5.2xlarge	8	52.8	Up to 10 Gigabit	0.961
cache.r5.4xlarge	16	105.8	Up to 10 Gigabit	1.921
cache.r5.12xlarge	48	317.8	10 Gigabit	5.775
cache.r5.24xlarge	96	635.6	25 Gigabit	11.550

C.2 Lambda pricing

Table C.3 lists the current pricing schema for AWS Lambda, as per April 2019 [10].

Table C.3: Amazon Lambda pricing (April 2019). Prices are in millionths of a US Dollar per 100ms of billed running time.

Memory (MB)	Price per 100ms	Memory (MB)	Price per 100ms	Memory (MB)	Price per 100ms
128	0.208	1088	1.771	2112	3.438
192	0.313	1152	1.875	2176	3.542
256	0.417	1216	1.980	2240	3.647
320	0.521	1280	2.084	2304	3.751
384	0.625	1344	2.188	2368	3.855
448	0.729	1408	2.292	2432	3.959
512	0.834	1472	2.396	2496	4.063
576	0.938	1536	2.501	2560	4.168
640	1.042	1600	2.605	2624	4.272
704	1.146	1664	2.709	2688	4.376
768	1.250	1728	2.813	2752	4.480
832	1.354	1792	2.917	2816	4.584
896	1.459	1856	3.021	2880	4.688
960	1.563	1920	3.126	2944	4.793
1024	1.667	1984	3.230	3008	4.897
		2048	3.334		

Bibliography

[1] Microsoft Security Response Center (MSRC). Accessed on 2019-04-07. June 2014. URL: https://blogs.technet.microsoft.com/msrc/2014/06/23/driving-a-collectively-stronger-security-community-with-microsoft-interflow/.

[2] Gojko Adzic and Robert Chatley. "Serverless computing: economic and architectural impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 884–889.

[3] Victor Manuel Alvarez. *YARA – The pattern matching swiss knife for malware researchers*. Accessed on 2019-04-11. URL: https://virustotal.github.io/yara/.

[4] Amazon Web Services, Inc. *Amazon ElastiCache pricing*. Accessed on 2019-04-12. URL: https://aws.amazon.com/elasticache/pricing/.

[5] Amazon Web Services, Inc. *Amazon Neptune Generally Available*. Accessed on 2019-05-12. May 2018. URL: https://aws.amazon.com/blogs/aws/amazon-neptune-generally-available/.

[6] Amazon Web Services, Inc. *Available CloudWatch Metrics for Amazon SQS*. Accessed on 2019-05-11. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-available-cloudwatch-metrics.html.

[7] Amazon Web Services, Inc. *AWS Lambda enables functions that can run up to 15 minutes*. Accessed on 2019-04-14. Oct. 2018. URL: https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/.

[8] Amazon Web Services, Inc. *AWS Lambda Limits*. Accessed on 2019-04-14. URL: https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

[9] Amazon Web Services, Inc. *AWS Lambda Metrics*. Accessed on 2019-05-11. URL: https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-metrics.html.

[10] Amazon Web Services, Inc. *AWS Lambda pricing*. Accessed on 2019-04-12. URL: https://aws.amazon.com/lambda/pricing/.

[11] Amazon Web Services, Inc. *Cloud Products*. Accessed on 2019-04-22. URL: https://aws.amazon.com/products/.

[12] Amazon Web Services, Inc. *Metrics for Redis*. Accessed on 2019-05-11. URL: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/CacheMetrics.Redis.html.

[13] Amazon Web Services, Inc. *Release: AWS Lambda on 2014-11-13*. Accessed on 2019-04-14. Nov. 2014. URL: https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/.

[14] Anti-Malware Testing Standards Organization, Inc. *The Use and Misuse of Test Files in Anti-Malware Testing*. 2016. URL: https://www.amtso.org/documents/.

[15] Apache Software Foundation. *Apache Spark documentation: Classification and Regression*. Accessed on 2019-06-14. URL: https://spark.apache.org/docs/latest/ml-classification-regression.html.

[16] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[17] Jean Camp et al. *Data for Cybersecurity Research: Process and "Wish List"*. 2010. URL: http://www.ljean.com/files/data-wishlist.pdf.

[18] Center for Applied Internet Data Analysis, based at the University of California's San Diego Supercomputer Center. *Internet Measurement Data Catalog (IMDC)*. Accessed on 2019-05-16. URL: http://www.caida.org/projects/trends/imdc/.

[19] Mandy Chessell et al. "Governing and managing big data for analytics and decision makers". In: *IBM Redguides for Business Leaders* (2014).

[20] Yun Chi et al. "Catch the moment: maintaining closed frequent itemsets over a data stream sliding window". In: *Knowledge and Information Systems* 10.3 (2006), pp. 265–294.

[21] Chronicle. *VirusTotal.com*. Accessed on 2019-04-10. URL: https://www.virustotal.com/.

[22] Catalin Cimpanu. "Online stores for governments and multinationals hacked via new security flaw". In: *ZDNet – Zero Day blog* (Jan. 2019). Accessed on 2019-05-14. URL: https://www.zdnet.com/article/online-stores-for-governments-and-multinationals-hacked-via-new-security-flaw/.

[23] OASIS Cyber Threat Intelligence (CTI) Technical Committee. *STIX™ Version 2.0., Committee Specification 01: Part 1: STIX Core Concepts*. Accessed on 2019-05-21. July 2017. URL: https://docs.oasis-open.org/cti/stix/v2.0/stix-v2.0-part1-stix-core.pdf.

[24] OASIS Cyber Threat Intelligence (CTI) Technical Committee. *TAXII™ Version 2.1. Committee Specification Draft 02 / Public Review Draft 01*. Accessed on 2019-05-21. Dec. 2018. URL: https://docs.oasis-open.org/cti/stix/v2.0/stix-v2.0-part1-stix-core.pdf.

[25] Association for Computing Machinery (ACM) SIGCOMM. *Internet Traffic Archive*. Accessed on 2019-05-16. URL: http://beta.sigcomm.org/ITA.

[26] F-Secure Corporation. *F-Secure Security Cloud: Purpose, Function and Benefits*. Accessed on 2019-04-09. Oct. 2015. URL: https://www.f-secure.com/documents/996508/1030745/security_cloud.pdf.

[27] F-Secure Corporation. *Security Cloud privacy policy*. Accessed on 2019-05-11. URL: https://www.f-secure.com/en/web/legal/privacy/security-cloud.

[28] F-Secure Corporation. *What is F-Secure Rapid Detection and Response Service (RDS)*. Last updated 2018-11-02. Accessed on 2019-04-09. Nov. 2018. URL: https://community.f-secure.com/t5/Business/What-is-F-Secure-Rapid-Detection/ta-p/112058.

[29] Luc Dandurand and Oscar Serrano Serrano. "Towards improved cyber security information sharing". In: *2013 5th International Conference on Cyber Conflict (CYCON 2013)*. IEEE. 2013, pp. 1–16.

[30] Antonella Di Stefano, Francesca Gangemi, and Corrado Santoro. "ERESYE: Artificial Intelligence in Erlang Programs". In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*. ERLANG '05. Tallinn, Estonia: ACM, 2005, pp. 62–71. ISBN: 1-59593-066-3. DOI: 10.1145/1088361.1088373.

[31] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[32] Tudor Dumitraş and Petros Efstathopoulos. "The provenance of wine". In: *2012 Ninth European Dependable Computing Conference*. IEEE. 2012, pp. 126–131.

[33] Tudor Dumitraş and Darren Shou. "Toward a Standard Benchmark for Computer Security Research: The Worldwide Intelligence Network Environment (WINE)". In: *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. BADGERS '11. Salzburg, Austria: ACM, 2011, pp. 89–96. ISBN: 978-1-4503-0768-0. DOI: 10.1145/1978672.1978683.

[34] *F-Secure Case Study: F-Secure Increases Customer Insight and Speeds Up Activation Using AWS*. Accessed on 2019-04-24. URL: https://aws.amazon.com/solutions/case-studies/f-secure/.

[35] Farsight Security, Inc. *Farsight Security Archive*. Accessed on 2019-05-16. URL: https://archive.farsightsecurity.com/.

[36] Farsight Security, Inc. *Security Information Exchange (SIE) protects from cybercrime*. Accessed on 2019-05-16. URL: https://www.farsightsecurity.com/solutions/security-information-exchange/.

[37] Ewan Gardner. "Serious Vulnerability Discovered in Adminer database Administration Tool". In: *Foregenix* (Jan. 2019). Accessed on 2019-05-14. URL: https://www.foregenix.com/blog/serious-vulnerability-discovered-in-adminer-tool.

[38] AV-TEST GmbH. *AV-TEST | Antivirus & Security Software & AntiMalware Reviews*. Accessed on 2019-04-10. URL: https://www.av-test.org/en/.

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[40] Cristin Goodwin and J Paul Nicholas. "A framework for cybersecurity information sharing and risk reduction". In: *Microsoft Research* (2015).

[41] Aryeh Goretsky. *Problematic, Unloved and Argumentative: What is a potentially unwanted application (PUA)?* Accessed on 2019-05-07. ESET, spol. s r.o., Nov. 2011. URL: https://www.welivesecurity.com/media_files/white-papers/Problematic-Unloved-Argumentative.pdf.

[42] Rihan Hai, Sandra Geisler, and Christoph Quix. "Constance: An intelligent data lake system". In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 2097–2100.

[43] *IMPACT*. Accessed on 2019-05-16. URL: https://www.impactcybertrust.org/.

[44] Hai Jin et al. "Cloud Types and Services". In: *Handbook of Cloud Computing*. Ed. by Borko Furht and Armando Escalante. Boston, MA: Springer US, 2010, pp. 335–355. ISBN: 978-1-4419-6524-0. DOI: 10.1007/978-1-4419-6524-0_14.

[45] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *arXiv preprint arXiv:1902.03383* (2019).

[46] Panos Kampanakis. "Security automation and threat information-sharing options". In: *IEEE Security & Privacy* 12.5 (2014), pp. 42–51.

[47] Stefan Katzenbeisser, Johannes Kinder, and Helmut Veith. "Malware Detection". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 752–755. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_838.

[48] Su Myeon Kim and Marcel-Catalin Rosu. "A Survey of Public Web Services". In: *E-Commerce and Web Technologies*. Ed. by Kurt Bauknecht, Martin Bichler, and Birgit Pröll. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 96–105. ISBN: 978-3-540-30077-9.

[49] Loren Kohnfelder and Praerit Garg. "The threats to our products". In: *Microsoft Interface, Microsoft Corporation* (1999), p. 33.

[50] Redis Labs. *An introduction to Redis data types and abstractions*. Accessed on 2019-04-08. URL: https://redis.io/topics/data-types-intro.

[51] *Letter symbols to be used in electrical technology – Part 2: Telecommunications and electronics*. IEC 60027-2. International Electrotechnical Commission, 1999.

[52] Jason Long. *JSON Lines, Documentation for the JSON Lines text file format*. Accessed on 2019-05-05. URL: http://jsonlines.org/.

[53] *Malicious URLs Tracking and Exchange group*. Accessed on 2019-04-10. URL: https://www.mutegroup.org/.

[54] Florian Menges and Günther Pernul. "A comparative analysis of incident reporting formats". In: *Computers & Security* 73 (2018), pp. 87–101.

[55] Natalia Miloslavskaya and Alexander Tolstoy. "Big data, fast data and data lake concepts". In: *Procedia Computer Science* 88 (2016), pp. 300–305.

[56] Jordan Novet. "Microsoft narrows Amazon's lead in cloud, but the gap remains large". In: *CNBC* (Apr. 2018). Accessed on 2019-04-14. URL: https://www.cnbc.com/2018/04/27/microsoft-gains-cloud-market-share-in-q1-but-aws-still-dominates.html.

[57] LLC OpenDNS. *PhishTank: An anti-phishing site*. Accessed on 2019-04-10. URL: https://www.phishtank.com.

[58] *OpenPhish – Phishing Intelligence*. Accessed on 2019-04-10. URL: https://openphish.com/.

[59] *Quantities and units – Part 13: Information science and technology*. ISO/IEC 80000-13. International Organization for Standardization / International Electrotechnical Commission, 2009.

[60] Claude Sammut and Geoffrey I. Webb, eds. *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer US, 2017. ISBN: 978-1-4899-7687-1. DOI: 10.1007/978-1-4899-7687-1_803.

[61] Clemens Sauerwein et al. "Threat intelligence sharing platforms: An exploratory study of software vendors and research perspectives". In: (2017).

[62] Nicolas Serrano, Gorka Gallardo, and Josune Hernantes. "Infrastructure as a service and cloud technologies". In: *IEEE Software* 32.2 (2015), pp. 30–36.

[63] Colleen Shannon et al. "The internet measurement data catalog". In: *ACM SIGCOMM Computer Communication Review (CCR)* 35.5 (2005).

[64] Adam Shostack. "Experiences Threat Modeling at Microsoft". In: *MODSEC@ MoDELS*. 2008.

[65] Ignacio G Terrizzano et al. "Data Wrangling: The Challenging Yourney from the Wild to the Lake". In: *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*. Jan. 2015.

[66] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[67] Peter Torr. "Demystifying the threat modeling process". In: *IEEE Security & Privacy* 3.5 (2005), pp. 66–70.

[68] U.S. Department of Homeland Security, Science and Technology Directorate. *Information Marketplace for Policy and Analysis of Cyber-risk & Trust*. Accessed on 2019-05-16. URL: https://www.dhs.gov/science-and-technology/cybersecurity-impact.

[69] U.S. Department of Homeland Security, Science and Technology Directorate. *Protected Repository for the Defense of Infrastructure Against Cyber Threats*. 2008. URL: https://www.archives.gov/files/records-mgmt/rcs/schedules/departments/department-of-homeland-security/rg-0563/n1-563-08-037_sf115.pdf.

[70] Koen Van Impe. "Signature-Based Detection With YARA". In: *SecurityIntelligence* (June 2015). Accessed on 2019-04-11. URL: https://securityintelligence.com/signature-based-detection-with-yara/.

[71] Andy Warzon. "Is AWS Fargate "Serverless"?" In: *Trek10* (May 2018). Accessed on 2019-05-03. URL: https://www.trek10.com/blog/is-fargate-serverless/.